Prof. Luiz Gustavo Turatti (PED-A)
Prof. Felipe B. Valio (PED-C)
Monitor Júlio César F. Cornacchia (PAD)

Versão 1.0.0 b – 05.out.2009

Índice

MC102 – Algoritmos e Programação de Computadores – Turma: Z – 2009s2

Página da disciplina: http://www.mc102.cjb.net

Critérios do Curso

Aulas

Dia	Horário	Sala	Conteúdo
Terça	21h00 as 22h40	CB03	Teórico
Quinta	19h00 as 20h45	CB04	Teórico
Sexta	21h00 as 22h40	LM03	Prático

Avaliações

18/09/2009 Prática (L1)
22/09/2009 Teórica (P1)
23/10/2009 Prática (L2)
27/10/2009 Teórica (P2)
24/11/2009 Teórica (P3)
27/11/2009 Prática (L3)
14/07/2009 Exame Final

Programa do Curso (ementa)

Tópicos a serem discutidos durante o semestre:

1. Introdução à Computação;
2. Algoritmos e Programas;
3. Variáveis e Atribuições;
4. Comandos Condicionais;
5. Comandos de Entrada e Saída;
6. Comandos de Repetição;
7. Vetores e Matrizes;
8. Cadeia de caracteres (strings);
9. Funções iterativas e recursivas;
10. Registros/Estruturas;
11. Listas ligadas;
12. Arquivo texto e binário;
13. Outros Tópicos.

Neste semestre será utilizada a linguagem de programação C (IDE DevC++) em ambiente Windows.

Atendimento

Ocorrerá logo após o término de cada aula e eventualmente através de e-mail para esclarecimento de dúvidas.

Exercícios

Durante o curso serão indicados vários exercícios práticos e teóricos. Não será exigida a entrega de todos, mas os conhecimentos adquiridos durante a resolução deles, que será cobrada nas avaliações.

Freqüência

O limite de faltas é de 25% do total das aulas previstas. Faltas não podem ser abonadas sob hipótese alguma (legislação do MEC). Única Exceção: Caso alguém tenha problemas de saúde que impeçam o seu comparecimento às aulas por períodos mais longos, então o aluno deve entrar com um requerimento de 'regime especial' junto à Diretoria Acadêmica (DAC). Isto pode ser feito por terceiros e, após a entrada de tal pedido, o aluno tem o direito de realizar as provas em casa enquanto estiver convalescendo.

A frequência às aulas teóricas e sessões de laboratório será atestada mediante assinatura de uma lista de presença e/ou chamada realizada pelo docente. A assinatura em listas de presença deve ser a oficial, isto é, a mesma usada em documentos formais depositados junto à DAC.

Observações Importantes

1. As avaliações e o exame final serão realizados nos horários correspondentes as aulas;
2. Qualquer tentativa de fraude nas avaliações implicará em aproveitamento ZERO na atividade fraudada para todos os envolvidos;
3. Não haverá atendimento extra em véspera de avaliações;
4. Não haverá avaliação substitutiva;
5. Não fará exame o aluno que não tiver a frequência mínima exigida.

Critério de pontuação das avaliações

Serão aplicadas provas teóricas (P1, P2 e P3) e as atividades em laboratório serão avaliadas pela participação do aluno durante as aulas + avaliações práticas (L1, L2 e L3).

O aproveitamento das avaliações teóricas será: P=(2*P1+3*P2+5*P3)/10

O aproveitamento das atividades em laboratório será: L=(2*L1+3*L2+5*L3)/10

O aproveitamento final do semestre será dado por: A=(6*P+4*L)/10

Onde:

P>=5.0 e L>=5.0 faz aluno aprovado;

P>=3.0 e L>=5.0 faz aluno em exame;

A média final é calculada por:

Se A>=5.0; M=A;

Senão M=(A+Exame)/2

Bibliografia Sugerida

Existem inúmeros textos sobre a linguagem C e seus variantes como Turbo C. Não será indicado qualquer texto em especial. As referências citadas são sugestões para seu estudo.

1. MEIRELLES, Fernando de Souza. "Informática: Novas aplicações com microcomputadores"; 2a Ed. São Paulo: Makron Books, 1994.
2. FORBELLONE, André L.V."Lógica de Programação - A construção de Algoritmos e estruturas de dados"; São Paulo: Makron Books, 1993.
3. GUIMARAES, Angelo de Moura. LAGES, Newton A.C."Algoritmos e estruturas de dados". Rio de Janeiro: LTC, 1985.
4. KERNIGHAN, B.W. RITCHIE, D.M. "The C Programming Language – ANSI C". Prentice Hall, 2nd Edition, 1978. ISBN: 978-0131101630.
5. SCHILDT, Herbert. "C Completo e Total". 3a Ed. Makron Books, 1997. ISBN: 8534605955.
6. GOTTFRIED, Byron S. "Programando em C – Coleção Schaum", MCGraw-Hill, 1993.
7. HARBISON, Samuel P. et al."C: A Reference Manual". 5a Ed. Prentice Hall, 2002. ISBN:0-13-089592X.
8. DEITEL,H.M. & DEITEL, P.J. "C How to Program - Introducing C++ and Java". 3a Ed. Prentice Hall, 2000.
9. PLANTZ,A.C. "C: Quick Reference". Que Pub, 1988. ISBN:9780880223720.
10. KOENIG, Andrew. "C Traps and Pitfalls". Addison-Wesley, 1989. ISBN:0-201-17928-8.
11. OUALLINE, Steve. "Practical C programming". 3rd Ed. O'Reilly,1997. ISBN:9781565923065.
12. HANSEN, Augie. "C Programming: A Complete Guide to Mastering the C Language".Addison-Wesley, 1989.
13. SCHILDT, Herbert. "C Avançado - Guia do Usuário". 2a Ed. McGraw-Hill, 1989.

Calendário do curso

AGOSTO
18/08 Aula inaugural: Introdução à computação
20/08 Introdução a lógica de programação
21/08 Aula em laboratório
25/08 Fluxogramas; Algoritmos; Exercícios de lógica
27/08 Fluxogramas; Algoritmos; Exercícios de lógica
28/08 Aula em laboratório

SETEMBRO
01/09 Introdução à linguagem C
03/09 Comandos de escrita, leitura e linearização de fórmulas
04/09 Aula em laboratório
08/09 Operadores; Estruturas de decisão, Exercícios
10/09 Operadores; Estruturas de decisão, Exercícios
11/09 Aula em laboratório
15/09 Estruturas de repetição, Exercícios
17/09 Estruturas de repetição, Exercícios
18/09 L1: Avaliação Prática (LM03)
22/09 P1: Avaliação Teórica (CB03)
24/09 Vetores
25/09 Aula em laboratório
29/09 Ordenação de vetores

OUTUBRO
01/10 Último dia para desistência de matrícula
01/10 Vetores multidimensionais (Matrizes)
02/10 Aula em laboratório
06/10 Cadeia de caracteres (strings) e Conversões
08/10 Ponteiros
09/10 Aula em laboratório
13/10 Ponteiros, Vetores e Strings
15/10 Funções
16/10 Aula em laboratório
20/10 Hierarquia de Funções
22/10 Recursão
23/10 L2: Avaliação Prática (LM03)
27/10 P2: Avaliação Teórica (CB03)
29/10 Recursão
30/10 Aula em laboratório

NOVEMBRO
03/11 Ordenação com recursão
05/11 Registros
06/11 Aula em laboratório
10/11 Listas
12/11 Listas
13/11 Aula em laboratório
17/11 Arquivos
19/11 Arquivos
20/11 Feriado Municipal
24/11 P3: Avaliação Teórica (CB03)
26/11 Sistemas
27/11 L3: Avaliação Prática (LM03)

DEZEMBRO
30/11 a 05/12: Semana de estudos
10/12 Exame
14/12 Divulgação da situação final

1 – Introdução à Computação

1.1 Motivação: Jamais desista!

A motivação deve ser vista como superação de quaisquer dificuldades que venham a surgir durante o curso.

http://www.youtube.com/watch?v=FB_pdyVc1bM

1.2 Evolução da Informática

A evolução do processamento de dados é o esforço do homem para encontrar meios melhores e eficientes de reunir dados de utilidade em sua vida, à medida que esses problemas aumentaram tanto em dimensões como complexidade.

O termo 'informática' foi criado na França em 1962 ('informatique') e provém da contração das palavras 'information automatique' (Informação automática).

A informática surgiu da idéia de auxiliar o homem nos trabalhos rotineiros e repetitivos, em geral, de cálculo e gerenciamento. Podemos definir informática então, como sendo a ciência que trata da informação.

1.2.1 Principais funções da informática

- Desenvolvimento de máquinas;
- Desenvolvimento de novos métodos de trabalho;
- Construção de aplicações automáticas;
- Melhoria de métodos e aplicações existentes.

1.2.2 Definição de computador

"aparelho concebido para desempenhar cálculos e operações lógicas com facilidade, rapidez e confiabilidade, segundo instruções (programas) nele introduzidas, constituído, de um modo geral, por unidade(s) de entrada, unidade de processamento central (C.P.U.), unidade de armazenamento principal permanente, memória temporária e unidade(s) de saída."

1.2.3 Histórico dos computadores

Relação histórica da máquina de calcular ao computador pessoal:

- 3500 A.C. – Ábaco: Egito; Dispositivo manual de cálculo
- 2600 A.C. – Ábaco: China (posteriormente chamado de Suan-Pan); Depois apareceu no Japão e foi chamado de Soroban. Até o século XVII era o mais rápido método de calcular.
- 1500 D.C. - Calculadora mecânica de Leonardo da Vinci
- 1614 - Logaritmos; John Napier; Ossos de Napier
- 1621 - Régua de cálculo; Willian Oughtred
- 1642 - Máquina aritmética; Blaise Pascal (Pascalina)
- 1672 - Calculadora universal; Gottfried Wilhelm Von Leibniz
- 1822 - Máquina das diferenças de Charles Babbage (somente projetada)
- 1833 - Máquina analítica: Charles Babbage e Ada Augusta Byron
- 1895 - Máquina de Herman Hollerith

- ~1900 - Máquina de Turing; Alan M. Turing. Teoria: Algoritmo como representação formal e sistemática de um processo; Um problema só terá solução algorítmica se existir uma Máquina de Turing que possa representá-lo.
- 1938 - Z1 Computador Eletromecânico; Konrad Zuse
- 1940 - Z2 Computador Eletromecânico; Konrad Zuse; Compostos por enorme quantidade de relés e circuitos. Entrada de dados através de filmes de 35 mm perfurados. Construído para ajudar Zuse nos cálculos em engenharia.
- 1943 - Z3 Computador Eletromecânico; Konrad Zuse
- 1944 - MARK-1; Howard Aiken; Características: Cartões perfurados; Manipulava até 23 dígitos; Obsoleto por ser baseado em relés.
- 1945 - 1º Bug de computador.
- 1946 - Computador Eletrônico: ENIAC (Eletronic Numerical Integrator and Calculator). Características: 30 toneladas; 19.000 válvulas eletrônicas; 1.500 relés; 200 kilowatts de consumo.
- 1947 - MARK-2
- 1949 - MARK-3; Já com sistema de programa armazenado.
- 1950 - Z4 Computador Eletromecânico; Konrad Zuse
- 1951 - Computador Eletrônico: UNIVAC-I (Universal Automatic Computer); Primeiro computador destinado ao uso comercial; Recebia instruções de uma fita magnética.
- 1954 - Computador Transistorizado; TX-0 (Transistorized eXperimental).
- 1961 - Surgimento do circuito integrado em escala comercial.
- 1971 - Surgimento do microprocessador; Uso comercial, Intel 4004; Primeiro computador pessoal.
- 1975 - Primeiro microcomputador produzido industrialmente; Altair 8800 utilizava processador Intel 8080.
- 1976 - 1º micro com sucesso comercial; Apple I criado por Steve Wozniak; Vários fabricantes entram neste mercado em franca expansão.
- 1980 - Seagate lança primeiro disco rígido com 5 MB.
- 1981 - IBM lança seu computador pessoal o IBM-PC com MS-DOS; A concorrência começa a lançar os computadores IBM-PC compatíveis, pois a tecnologia estava aberta desde 1969.
- 1983 - Apple lança o Lisa, primeiro computador com interface gráfica
- 1984 - Apple lança o Macintosh (interface gráfica + mouse) para competir com o IBM-PC AT

1.2.4 Geração dos computadores

1ª geração (1940-1952)

- Computadores a base de válvulas a vácuo e relés
- Usados para aplicações científicas e militares
- Armazenamento de dados em cartões perfurados
- Programação na linguagem de máquina

2ª geração (1952-1964)

- Computadores baseados nos transistores
- Máquinas bem menores
- Aplicações científicas, militares, e agora também, administrativa e gerencial
- Primeiras linguagens de programação
- Fitas magnéticas e tambores magnéticos como memória

3ª geração (1964-1971)
- Computadores baseados em circuitos integrados
- Grande evolução dos Sistemas operacionais
- Memória em semicondutores e discos magnéticos
- Multiprogramação, Real time

4ª geração (1971-1981)
- Início com o surgimento do microprocessador
- Grande redução do tamanho dos computadores
- Surgem muitas linguagens de alto-nível
- Intel inaugura uma nova fase, oferecendo microcomputadores mais rápidos e que possibilitam a execução de várias tarefas ao mesmo tempo.

5ª geração (1981 até a atualidade)
- Processadores com altíssima velocidade de processamento
- Inteligência Artificial

1.2.5 Classificação dos computadores

ANTIGA	NOVA
Microcomputador	Portátil
Supermicro	Desktop
Minicomputador	Servidor de rede
Supermini	Grande Porte
Grande Porte (MainFrame)	Supercomputador
Supercomputador	

Sugestão complementar:
- http://www.computerhistory.org/timeline
- Filme: Piratas do Vale do Silício

1.3 Introdução à programação

Os computadores são dispositivos eletrônicos que transmitem, armazenam e manipulam informações (dados). As aplicações técnicas e científicas preocupam-se fundamentalmente com dados numéricos, enquanto aplicações comerciais normalmente envolvem processamento tanto numérico como de caracter. Para processar um conjunto particular de dados, o computador deve ter um conjunto de instruções apropriadas, denominado programa.

1.3.1 Tipos de linguagem de programação

Diferentes linguagens podem ser utilizadas para programar um computador. A mais básica é a linguagem de máquina (assembly language) que é composta por um conjunto de instruções detalhadas e obscuras que controlam um circuito interno do computador. Esse é o dialeto natural do computador. Na realidade, poucos programas são escritos em linguagem de máquina por duas razões importantes: primeiro porque a linguagem de máquina é muito trabalhosa, e segundo, porque a maioria dos equipamentos possui seu próprio conjunto de instruções. Isto torna a linguagem de máquina específica para cada arquitetura de computadores.

Normalmente um programa será escrito em alguma linguagem de alto nível, cujo conjunto de instruções é mais compatível com a linguagem humana e com o modo humano de pensar. A maior parte dessas linguagens é de uso genérico como BASIC, C, COBOL, FORTRAN, JAVA, PASCAL entre outras, além de linguagens de uso específico (LISP, por exemplo, utilizada na área de inteligência artificial).

Os programas escritos em linguagem de alto nível possuem instruções onde uma instrução equivalerá a várias instruções em linguagem de máquina. Além disso, as mesmas regras básicas de programação aplicam-se a todos os computadores. Portanto, um programa escrito em linguagem de alto nível geralmente pode ser executado em diferentes computadores com pouca ou nenhuma alteração. Por essas razões, o uso de linguagem de alto nível oferece três vantagens significativas sobre o uso de linguagem de máquina: simplicidade, uniformidade e portabilidade (independência da máquina).

Um programa escrito em linguagem de alto nível precisa ser traduzido para linguagem de máquina antes de ser executado. Essa tradução é conhecida como compilação ou interpretação, dependendo de como é executada. Compiladores traduzem o programa todo para a linguagem de máquina antes de executar qualquer instrução. Interpretadores, por outro lado, seguem através do programa traduzindo e, então, executando uma instrução isolada ou pequenos grupos de instrução. Em ambos os casos, a tradução é executada automaticamente pelo computador. Na verdade, programadores inexperientes podem até não perceber qual dos processos está acontecendo, já que normalmente vêem apenas o programa original em linguagem de alto nível, os dados de entrada e os dados de saída resultantes.

Um compilador ou interpretador não passa de um programa de computador que aceita um programa de alto nível (por exemplo, um programa em C) como dado de entrada e gera um programa correspondente em linguagem de máquina como saída. O programa de alto nível original é chamado de programa-fonte e o programa em linguagem de máquina resultante é o programa-objeto. Toda linguagem de alto nível precisa ter seu próprio compilador ou interpretador para um determinado computador. A maioria das implementações em C opera como compilador, apesar de interpretadores estarem se tornando cada vez mais comuns.

Em geral, é mais conveniente desenvolver um novo programa usando um interpretador do que um compilador. Uma vez que o programa não apresenta erros, uma versão compilada executará mais rapidamente que uma versão interpretada. As razões disso estão fora do alcance de nossa discussão.

1.3.2 O surgimento do UNIX

As raízes do UNIX datam de meados dos anos 60, quando a AT&T, Honeywell, GE e o MIT embarcaram em um massivo projeto para desenvolvimento de um utilitário de informação, chamado Multics (Multiplexed Information and Computing Service).

Multics era um sistema modular montado em uma bancada de processadores, memórias e equipamentos de comunicação de alta velocidade. Pelo desenho, partes do computador poderiam ser desligadas para manutenção sem que outras partes ou usuários fossem afetados. O objetivo era prover serviço 24 horas por dia 365 dias por ano - um computador que poderia ser tornado mais rápido adicionando mais partes.

Em 1969, o projeto estava muito atrasado em relação ao seu cronograma e a AT&T resolveu abandona-lo. O projeto continuou no MIT.

Neste mesmo ano, Ken Thompson, um pesquisador da AT&T que havia trabalhado no Projeto Multics, pegou um computador PDP-7 para pesquisar algumas idéias do Multics por conta própria. Logo Dennis Ritchie, que também trabalhou no Multics, se juntou a ele. Enquanto Multics tentava fazer várias coisas, UNIX tentava fazer uma coisa bem: rodar programas. Este pequeno escopo era todo ímpeto que os pesquisadores precisavam.

1.3.3 A evolução das linguagens de programação

A linguagem Algol foi definida por um comitê de cientistas europeus em 1960, mas não surgiram compiladores para ela de imediato. Na tentativa de ter um Algol simplificado, pesquisadores da Universidade de Cambridge (Inglaterra) desenvolveram a linguagem BCPL. A partir dela, Dennis Ritchie desenvolveu uma linguagem a qual chamou de B, com o objetivo de poder escrever compiladores portáteis. Thompson e Ritchie unificaram seus esforços, levando a uma primeira definição da linguagem C por volta de 1971.

1.3.4 A evolução da linguagem C

A partir de 1971 o UNIX começou a incorporar as características que possui atualmente e em 1973 foi completamente reescrito em linguagem C. Inicia-se então a grande sinergia entre UNIX a linguagem C. Desta evolução, podemos acompanhar a evolução do UNIX e suas derivações.

Em 1978, Ritchie e Thompson publicaram "C – The Programming Language". Este livro tornou-se um clássico por conter uma primeira definição da linguagem C disponível fora do Bell Labs.

1.3.5 A padronização da linguagem C

Até 1983, a maioria dos programas eram escritos em assembler (linguagem específica para cada tipo de equipamento) e a partir desse ano houve maior adesão a linguagem C. A padronização foi constituída pelo American National Standards Institute (ANSI), um comitê com o objetivo de padronizar a linguagem C em virtude da pressão econômica criada sobre ela, onde a definição original passou a ser insuficiente. Desde então diversas propostas foram feitas (IBM, Lattice, Microsoft, Borland) para aceitação gradual do padrão.

1.3.6 Porque estudar a linguagem C?

Entre muitas razões da popularidade desta linguagem, destacamos:

- Linguagem de programação de alto nível, estruturada e flexível;
- Inclui certas características de baixo nível que normalmente estão disponíveis somente em assembler (linguagem de máquina);
- Os programas escritos em C geram, depois de compilados, programas-objetos pequenos e eficientes;
- É uma linguagem amplamente disponível, desde computadores pessoais a mainframes;
- É independente da máquina, permitindo que os programas sejam facilmente transportados de um computador para outro.

1.3.7 O surgimento do Linux

Em 1983, Richard Matthew Stallman iniciou o movimento que deu origem a Free Software Foundation (GNU Project) fundada em 1985 (www.fsf.org) cujo objetivo foi prover um substituto ao sistema operacional UNIX (que era distribuído sob licença de uso comercial).

Andrew Stuart Tanenbaum é o autor do MINIX (mini-UNIX), um sistema operacional baseado no UNIX versão 7 (1977), com propósito educacional. Segue o padrão POSIX (Portable Operating System Interface), que garante portabilidade do código entre sistemas. A idéia foi criar um UNIX com micronúcleo para oferecer a funcionalidade mínima no núcleo para torná-lo confiável e eficiente.

Descontente com o MINIX, em 1991, Linus Benedict Torvals iniciou o desenvolvimento de seu próprio kernel (núcleo do sistema operacional) para oferecer uma solução melhor que o MINIX (www.minix3.org). Mal sabia Torvalds que seu passatempo tomaria as dimensões atuais. A primeira versão estável foi disponibilizada em 1994 (v 1.0) e atualmente encontra-se na versão 2.6.28.5 (12 de fevereiro de 2009).

1.3.8 Porque Linux?

Baseado nos argumentos apresentados anteriormente, o GNU/Linux é um sistema operacional estável, livre e com ótimo desempenho. Tem suas raízes no meio acadêmico e oferece inúmeras qualidades e possibilidades de adequação às necessidades de seus usuários. Dentre elas podemos destacar:

- Estabilidade: sem telas azuis, travamentos e problemas virais freqüentes;
- Livre e aberto: liberdade no sentido de permitir ao utilizador adequar o sistema a suas necessidades, seja na instalação de aplicativos gratuitos, interface, segurança e otimização do sistema entre outras possibilidades;
- Ambiente rico em ferramentas para estudo e desenvolvimento de aplicações;
- Alto desempenho em redes de computadores (Intranet e Internet);
- Apoio do MEC (Ministério da Educação – Brasil) através do ProInfo.

1.4 Organização do computador

O computador é uma máquina que funciona mediante instruções enviadas pelo ser humano ou por outra máquina, executando tarefas e resolvendo problemas tais como: cálculos complexos, geração de relatórios, comando de outras máquinas (um robô, por exemplo), controle de contas bancárias, comunicação de informações, entre outras. Uma visão simplificada do computador é ilustrada na figura 1.

Unidades de entrada: Usadas pelo computador para receber instruções ou informações externas. Exemplo: teclado, mouse, microfone, câmera de vídeo, etc.

Unidades de saída: Usadas pelo computador para exibir os resultados da computação. Exemplo: monitor, impressora, etc.

Unidade Central de Processamento (**CPU** = Central Processing Unit): Responsável pelo gerenciamento do sistema como um todo, incluindo as memórias e as unidades de entrada e saída.

Unidade Lógica e Aritmética (ULA): Responsável pelos cálculos matemáticos. Alguns computadores tem esta unidade separada da CPU. Também chamado de co-processador matemático.

Memória Principal (ou memória primária): Usada pela CPU para armazenar instruções e informações enquanto o computador está ligado. Também conhecida como memória RAM (Random Access Memory).

Memória Secundária: Usada pelo computador para armazenar instruções e informações por prazo indeterminado, independente do estado do computador (ligado ou desligado). Em geral com capacidade de armazenamento bem maior do que a memória RAM, mas de acesso mais lento. Exemplo: disco rígido (HDD; winchester), disquetes, fitas magnéticas, etc.

Observação: As memórias primária e secundária podem ser vistas como unidades de entrada e saída.

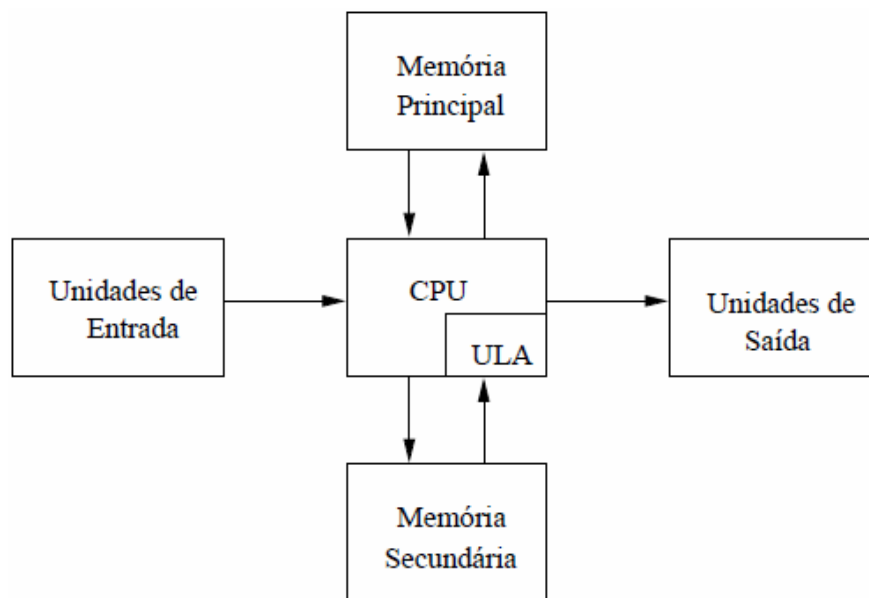



Figura 1: Organização básica de um computador.

1.5 Alguns termos técnicos

bit: unidade básica para armazenamento, processamento e comunicação de dados.

byte: conjunto de 8 bits.

dados: qualquer tipo de informação ou instrução que pode ser manipulada pelo computador. Exemplo: textos, imagens, etc.

palavra (word): é um conjunto de N bytes (ex: N = 1, 2, 4, 8). Os dados são organizados em palavras de N bytes. Os processadores mais modernos tratam os dados em palavras de 16 bytes ou 128 bytes.

comandos: são as instruções que fazem o computador executar tarefas.

programa: é uma seqüência de instruções com alguma finalidade.

arquivo: conjunto de bytes que contém dados. Estes dados podem ser: um programa, uma imagem, uma lista com nomes de pessoas, etc.

software: é um conjunto de programas com um propósito global em comum.

hardware: consiste na parte física do computador.

sistema operacional: conjunto de programas que gerenciam e alocam recursos de hardware e software. Exemplo: UNIX, MS-DOS, Windows, OS/2, MAC-OS, Linux, entre outros.

linguagem de programação: consiste da sintaxe (gramática) e semântica (significado) utilizada para escrever (ou codificar) um programa.

(a) Alto Nível: linguagem de codificação de programa independente do tipo de máquina e de fácil utilização pelo ser humano. Exemplo: Pascal, C, COBOL, BASIC, JAVA entre outras.

(b) Baixo Nível: linguagem de codificação baseada em mnemônicos. Dependente do tipo de máquina e de fácil tradução para a máquina. Conhecida como linguagem assembly.

linguagem de máquina: conjunto de códigos binários que são compreendidos pela CPU de um dado computador. Depende do tipo de máquina.

compilador: traduz programas codificados em linguagem de alto ou baixo nível (código fonte) para linguagem de máquina (código executável). O assembler transforma um programa em assembly para linguagem de máquina. Uma vez compilado, o programa pode ser executado em qualquer máquina com o mesmo sistema operacional para o qual o programa foi compilado.

interpretador: traduz o código fonte para código de máquina diretamente em tempo de execução. Exemplos de linguagens interpretadas são: BASIC, Python, TCL/TK e LISP.

algoritmos: são procedimentos ou instruções escritos em linguagem humana antes de serem codificados usando uma linguagem de programação. Uma receita de bolo é um bom exemplo da organização de um algoritmo.

1.6 Objetivos do curso

A figura 2 mostra um diagrama das tarefas básicas para a solução de problemas usando um computador. O objetivo principal deste curso é exercitar estas tarefas definindo vários conceitos de computação e usando a linguagem C como ferramenta de programação.

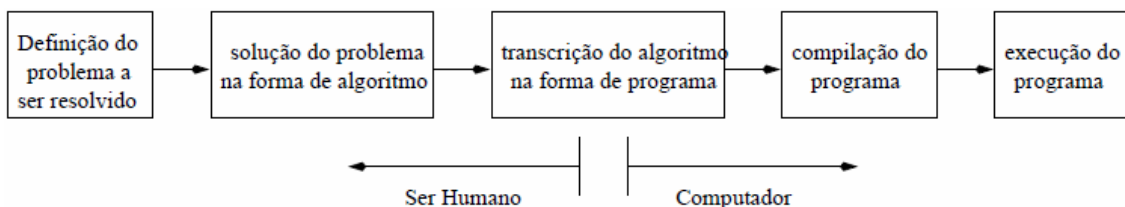



Figura 2: Etapas da resolução de problemas usando um computador.

1.7 Utilização da memória para programação

Essencialmente programar um computador para executar uma dada tarefa é estabelecer regras de manipulação de informações na sua memória principal através de uma seqüência de comandos. A memória principal funciona como um escaninho, cuja configuração varia de programa para programa. Cada programa estabelece o número de espaços no escaninho, onde cada espaço possui nome, endereço e capacidade de armazenamento, diferentes. Nesses espaços é possível armazenar números inteiros, números reais e caracteres, os quais requerem número de bytes diferentes. O conteúdo deste espaço pode ser lido ou modificado utilizando seu nome ou endereço.

Suponha, por exemplo, que gostaríamos que o computador calculasse a soma de dois números inteiros. Assim como o ser humano, os números são armazenados na memória, são somados e depois o resultado é armazenado na memória. Para armazenar os números na memória, precisamos estabelecer nome, endereço e capacidade de armazenamento de cada espaço compatível com um número inteiro. Depois atribuímos valores aos espaços do escaninho. Como os números não serão mais necessários após a soma, podemos usar algum dos espaços definidos para armazenar o resultado. Pedimos então que o computador execute a soma e armazene o resultado num

determinado espaço. A figura 3 ilustra a situação proposta acima. Os espaços no escaninho a que nos referenciamos são chamados de variáveis.

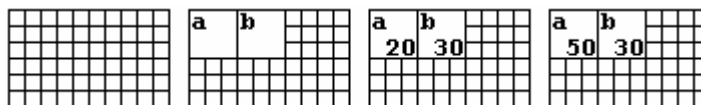



Figura 3: Exemplo de alocação de memória pelo programa

<table>
<tr><th>Algoritmo</th><th>Programa transcrito em linguagem C</th></tr>
<tr><td>Considere os espaços a e b;
Atribua 20 para a e 30 para b;
Some a com b e coloque o resultado em a;

ou seja

Sejam a e b variáveis inteiras;
Faça a←20 e b← 30;
Faça a ← a + b.</td><td><pre>/* Função principal. As funções e alguns
comandos tem escopo definido entre
parênteses */
#include <stdio.h>
int main()
{ int a, b;
   a=20; b=30;
   a=a+b;
   /* imprime resultado na tela */
   printf(“%d”, a);
   return(0);
}</pre></td></tr>
</table>

Em ambiente GNU/Linux, podemos utilizar o editor de texto **emacs** ou a interface ANJUTA para programação (http://projects.gnome.org/anjuta/index.shtml).

Para compilar o programa:

```
[usuario]$ gcc exemplo1.c –o exemplo1
```

Para executar o programa:

```
[usuario]$ ./exemplo1
```

Em ambiente Windows, podemos utilizar o Dev-C++ ou CodeBlocks:

- http://www.bloodshed.net/devcpp.html
- http://www.codeblocks.org/

Na primeira aula em laboratório utilizaremos Windows e Dev-C++ onde serão apresentadas as funcionalidades desta interface para desenvolvimento.

1.8 Exercícios: Laboratório 1

- Apresentação do laboratório;
- Criação de conta (usuário/senha) para os alunos;
- Apresentação do ambiente de trabalhos;
- Demonstração de exemplos;
- Demonstração de possíveis erros e como verificar o código.

Exemplo1: Bem vindos alunos da turma MC102

```
/* Primeiro Programa */
#include <stdio.h>

int main()
{
   printf(“\n\nBem-vindos alunos da turma MC102\n\n”);
   return 0;
}
```

2 – Lógica de Programação

2.1 Introdução à Lógica de Programação

2.1.1 Lógica

A lógica de programação é necessária para pessoas que desejam trabalhar com desenvolvimento de sistemas e programas, ela permite definir a seqüência lógica para o desenvolvimento.

Então o que é **lógica de programação**?

É a técnica de encadear pensamentos para atingir determinado objetivo.

Lógica é a arte de pensar corretamente. A lógica estuda a correção do raciocínio.

Ex. São Paulo é um estado do Brasil.
Campinas é uma cidade do estado de São Paulo.
Portanto, Campinas está localizada no Brasil.

2.1.2 Seqüência Lógica

Estes pensamentos, podem ser descritos como uma seqüência de instruções, que devem ser seguidas para se cumprir uma determinada tarefa.

Seqüência Lógica são passos executados até atingir um objetivo ou solução de um problema.

2.1.3 Exercícios que envolvem a lógica

2.1.3.1) Os jogadores

Quatro homens se reúnem diariamente para jogar e conversar. Não há dois com cabelos da mesma cor e que tenham preferência pelo mesmo jogo. Com as seguintes indicações dadas, você conseguiria completar o quadro associando à cor de cabelo e ao jogo preferido de cada um?

a) Vicente tem cabelos castanhos.
b) O jogador de pôquer é louro.
c) Alguém gosta de jogar dominó, mas não é o Roberto, pois ele gosta de jogar pôquer.
d) Lucas gosta de jogar damas.
e) O jogador de xadrez tem cabelos brancos.
f) Pedro não tem cabelos ruivos.

	Vicente	**Lucas**	**Pedro**	**Roberto**
Cabelos:				
Jogo:				

2.1.3.2) Amigos do Esporte

Quatro atletas tornaram-se amigos durante uma competição internacional. São eles: Paul, George, Luís e um atleta da Rússia. Com base nos dados abaixo, descubra o nome, o esporte e o país de cada atleta.

a) Dimitri não jogava futebol nem praticava atletismo.
b) O atleta dos EUA praticava basquete.
c) Luís não era dedicado à ginástica nem era norte-americano.
d) George não era nem do Brasil nem dos EUA.
e) Paul não jogava futebol nem praticava atletismo.
f) O atleta da Inglaterra não praticava nem ginástica nem futebol.

Atleta:	País:	Esporte:

2.1.3.3) Os cinco amigos

Um grupo de cinco amigos decidem ir a um Estádio de futebol, mas irão se encontrar lá dentro do Estádio. Para ficar mais fácil de se encontrarem, cada um vai com um boné de uma cor, e uma camisa de um time diferente, não há dois amigos com a mesma camisa nem com boné da mesma cor. Baseando nas afirmações abaixo, descubra quem está com que camisa e qual a cor do boné que cada um estava usando.

a) Marcos usa boné vermelho.
b) Robson está usando a camisa do Brasil.
c) O garoto que está com a camisa do São Paulo não usa boné amarelo.
d) Quem usa boné verde está com a camisa do Guarani.
e) Victor não usa boné amarelo.
f) André usa boné azul e não preto.
g) Victor está com a camisa do Santos.
h) O garoto de boné azul está com a camisa do Cruzeiro.

Nome:	Camisa:	Boné:
Antônio		
Robson		
Marcos		
Victor		
André		

2.1.3.4) F-1

O diretor da corrida de F-1 esqueceu qual é a ordem de largada da corrida, e necessita saber disso para informar os jornalistas. Sabendo que são apenas seis pilotos e que cada um pertence a uma equipe onde as cores dos carros são diferentes, ajude-o a formar a ordem de largada baseando-se nas seguintes afirmações:

a) O 1º colocado tem um carro amarelo.
b) Eddie, vai largar na frente da equipe Lola e atrás do carro azul.
c) O carro de Rosset é preto.
d) O último colocado está atrás da Lola.
e) Senna está atrás de um carro branco.
f) Se invertessem a ordem, a equipe Tasman ficaria em primeiro.

g) O carro da equipe Brabham é vermelho.
h) O carro do 5º colocado é preto.
i) Richie é o piloto da equipe Lotus.
j) O dono do carro verde é Luyendyk.
k) A equipe de Minardi está entre as equipes Lotus e March.
l) Patrese, Rosset, Luyendyk e Eddie não conseguiram ficar em 1° lugar.

Ordem:	Piloto:	Equipe:	Cor do carro:
1º			
2º			
3º			
4º			
5º			
6º			

2.1.4 Instruções

Na linguagem comum, entende-se por instruções “um conjunto de regras ou normas definidas para a realização ou emprego de algo”.

Em informática, porém, instrução é a informação que indica a um computador uma ação elementar a executar.

Convém ressaltar que uma ordem isolada não permite realizar o processo completo, para isso é necessário um conjunto de instruções colocadas em ordem seqüencial lógica.

Por exemplo, se quisermos fazer uma “Omelete de batatas”, precisaremos colocar em prática uma série de instruções: descascar as batatas, bater os ovos, fritar as batatas, etc. É evidente que estas instruções devem ser executadas em uma ordem adequada como não descascar as batatas depois de fritá-las, por exemplo.

Dessa maneira, uma instrução tomada em separado não tem muito sentido; para obtermos o resultado, precisamos colocar em prática o conjunto de todas as instruções, na ordem correta.

Instruções são um conjunto de regras ou normas definidas para a realização ou emprego de algo. Em informática, é o que indica a um computador uma ação elementar a executar.

2.1.5 Algoritmo

Um algoritmo é formalmente uma **seqüência finita de passos** que levam a execução de uma tarefa. Podemos pensar em algoritmo como uma receita, uma seqüência de instruções que dão cabo de uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devem ser claras e precisas.

Como exemplos de algoritmos podemos citar as operações básicas (multiplicação, divisão, adição e subtração) de números reais decimais. Outros exemplos seriam os manuais de aparelhos eletrônicos, que explicam passo-a-passo como executar uma determinada tarefa, por exemplo, converter uma música de um CD para MP3.

Até mesmo as coisas mais simples, podem ser descritas por seqüências lógicas.Por exemplo:

“Chupar uma bala”.

- Pegar a bala
- Retirar o papel
- Chupar a bala
- Jogar o papel no lixo

“Somar dois números inteiros quaisquer”.

- Escreva o primeiro número no retângulo A
- Escreva o segundo número no retângulo B
- Some o número do retângulo A com número do retângulo B e coloque o resultado no retângulo C

2.1.5.1 Algoritmizando a lógica

Algoritmo é uma seqüência de passos que visam atingir um objetivo bem definido.

Ex. Trocar uma lâmpada queimada.

```
Ligue o interruptor;
Se a lâmpada não acender, então:
      Pegue uma escada;
      Posicione-a embaixo da lâmpada;
      Busque uma lâmpada nova;
      Suba na escada;
      Retire a lâmpada velha;
      Coloque a lâmpada nova.
```

Aqui ainda não vamos entrar em detalhes de como proceder SE algo acontecer. Num primeiro momento, consideremos que determinada lâmpada está queimada. Então para trocá-la fazemos:

```
Pegue uma escada;
Posicione-a embaixo da lâmpada;
Busque uma lâmpada nova;
Suba na escada;
Retire a lâmpada velha;
Coloque a lâmpada nova;
Desça da escada;
Descarte a lâmpada queimada;
Guarde a escada.
```

2.1.5.2 Método para construção de algoritmos

a) Ler atentamente o enunciado.
b) Retirar do enunciado a relação das entradas de dados.
c) Retirar do enunciado a relação das saídas de dados.
d) Determinar o que deve ser feito para transformar as entradas determinadas nas saídas especificadas.
e) Construir o algoritmo.
f) Executar o algoritmo.

2.1.6 Programas

Os programas de computadores nada mais são do que algoritmos escritos numa linguagem de computador (Pascal, C, Cobol, Fortran, Visual Basic entre outras) e que são interpretados e executados por uma máquina, no caso um computador. Notem que dada esta interpretação rigorosa, um programa é por natureza muito específico e rígido em relação aos algoritmos da vida real.

2.1.7 Exercícios

Crie uma seqüência lógica (escreva um algoritmo), detalhando as etapas para:

2.1.7.1) Abrir uma porta com chave;
2.1.7.2) Tomar banho;
2.1.7.3) Trocar um pneu do carro;
2.1.7.4) Trocar uma lâmpada;
2.1.7.5) Somar dois números inteiros e multiplicar o resultado pelo primeiro número.

2.2 Desenvolvendo algoritmos

2.2.1 Pseudocódigo

Os algoritmos são descritos em uma linguagem chamada pseudocódigo. Este nome é uma alusão à posterior implementação em uma linguagem de programação, ou seja, quando formos programar em linguagem C, por exemplo, estaremos gerando código fonte nessa linguagem. Por isso os algoritmos são independentes das linguagens de programação. Ao contrário de uma linguagem de programação não existe um formalismo rígido de como deve ser escrito o algoritmo.

O algoritmo deve ser fácil de se interpretar e fácil de codificar. Ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação.

2.2.2 Regras para construção do Algoritmo

Para escrever um algoritmo precisamos descrever a seqüência de instruções, de maneira simples e objetiva. Para isso utilizaremos algumas técnicas:

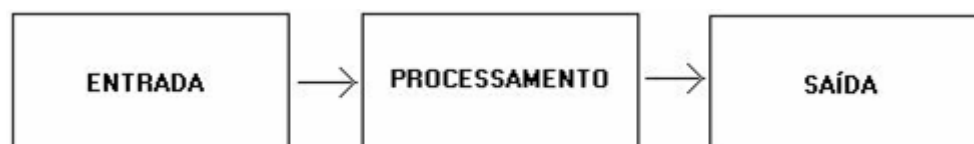
- Usar somente um verbo por frase
- Imaginar que você está desenvolvendo um algoritmo para pessoas que não trabalham com informática
- Usar frases curtas e simples
- Ser objetivo
- Procurar usar palavras que não tenham sentido dúbio

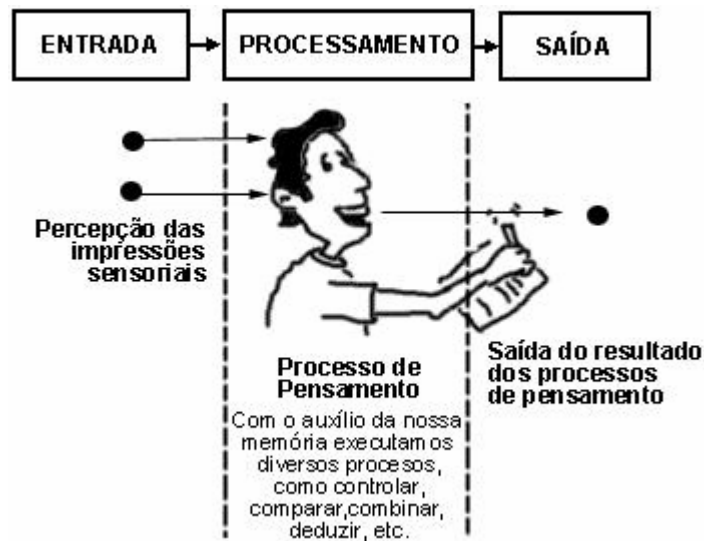
2.2.3 Fases

No capítulo anterior vimos que ALGORITMO é uma seqüência lógica de instruções que podem ser executadas.

É importante ressaltar que qualquer tarefa que siga determinado padrão pode ser descrita por um algoritmo, como por exemplo:

- COMO FAZER ARROZ DOCE
- CALCULAR O SALDO FINANCEIRO DE UM ESTOQUE

Entretanto ao montar um algoritmo, precisamos primeiro dividir o problema apresentado em três fases fundamentais. Onde temos:

- ENTRADA: São os dados de entrada do algoritmo;
- PROCESSAMENTO: São os procedimentos utilizados para chegar ao resultado final;
- SAÍDA: São os dados já processados.

2.2.4 Analogia com o homem

2.3 Exemplo de Algoritmo

Imagine o seguinte problema:

Calcular a média final dos alunos de uma determinada série. Os alunos realizarão quatro provas: P1, P2, P3 e P4 com valores reais (6,25 por exemplo). Onde:

Média Final = (P1 + P2 + P3 + P4)/4

Para montar o algoritmo proposto, faremos três perguntas:

a) Quais são os dados de entrada? R: São P1, P2, P3 e P4

b) Qual será o processamento a ser utilizado (o que fazer)? R: Somar entradas e dividir por quatro.

c) Quais serão os dados de saída? R: O dado de saída será a média final

Algoritmo

- Receba a nota da prova1
- Receba a nota de prova2
- Receba a nota de prova3
- Receba a nota da prova4
- Some todas as notas
- Divida o resultado por 4
- Mostre o resultado da divisão

2.4 Teste de Mesa

Após desenvolver um algoritmo ele deverá sempre ser testado. Este teste é chamado de TESTE DE MESA, que significa, seguir as instruções do algoritmo de maneira precisa para verificar se o procedimento utilizado está correto ou não. Utilize a tabela abaixo:

P1	P2	P3	P4	Média

2.5 Exercícios

2.5.1) Faça um algoritmo que calcule o valor total das peças, identificando os dados de entrada, processamento e saídas onde se deva:

- Receber o valor da peça
- Receber a quantidade de peças
- Calcular o valor total das peças (quantidade X valor)
- Mostrar o valor total

2.5.2) Faça um algoritmo para calcular o estoque médio de uma peça:

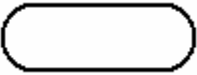
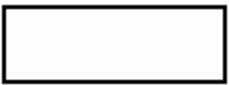
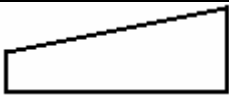
EstoqueMedio = (Quantidade MINima + Quantidade MAXima) / 2

2.5.3) Faça o teste de mesa para os algoritmos anteriores com dados definidos por você.

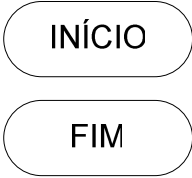
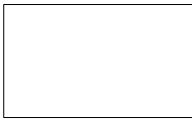
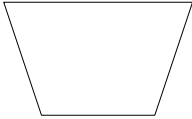
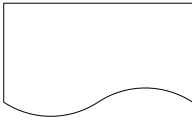
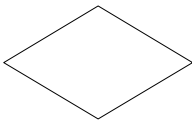
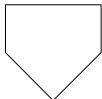
3 – Diagrama de Bloco (Fluxograma)

3.1 O que é um diagrama de bloco?

O diagrama de blocos é uma forma padronizada e eficaz para representar os passos lógicos de um determinado processamento. Com o diagrama podemos definir uma seqüência de símbolos, com significado bem definido, portanto, sua principal função é a de facilitar a visualização dos passos de um processamento.

O objetivo principal do fluxograma é descrever a seqüência (fluxo), seja manual ou mecanizado, especificando os suportes (documento, papel, mídia, formulário ou qualquer outro) que sejam usados para os dados e as informações.

Em sua confecção, são usados símbolos convencionados, que permitem poucas variações. Apresenta como principal característica ser claro e objetivo, sendo o mais utilizado de todos os instrumentos e ferramentas à disposição do analista, embora poucos profissionais o empreguem de forma pura.

O documento final elaborado deve estar constituído por três grandes partes integrantes, a saber:

- **Cabeçalho**: deve conter todas as informações necessárias para identificar claramente ao que se refere, incluindo nome do projeto e número de identificação (se houver), nome do (sub) sistema, nome do processo, data, quem elaborou e outras informações de identificação que sejam necessárias.
- **Corpo**: contém o fluxograma propriamente dito.
- **Explicação**: devem ser colocadas todas as explicações que se façam necessárias em consultas futuras, tais como: informações quantitativas (freqüência e volume); tempo total, desde a primeira entrada até o final; níveis de autoridade, quando alguma ação depende de aprovação ou confirmação por escrito; informações ou esclarecimentos adicionais.

3.2 Simbologia

Existem diversos símbolos em um diagrama de bloco. Os mais utilizados são:

Símbolo	Função
TERMINAL	Indica o INICIO ou FIM de um fluxo de dados
PROCESSAMENTO	Indica o processamento de informações Exemplo: SOMA = A + B
ENTRADA	Indica a entrada de dados através do teclado
SAÍDA	Mostra mensagens ao usuário: informações, solicitações ou resultados.
CONECTOR	Indica o sentido do fluxo de dados e conecta símbolos e/ou blocos.

SÍMBOLOS

Relação de alguns componentes utilizados em fluxogramas:

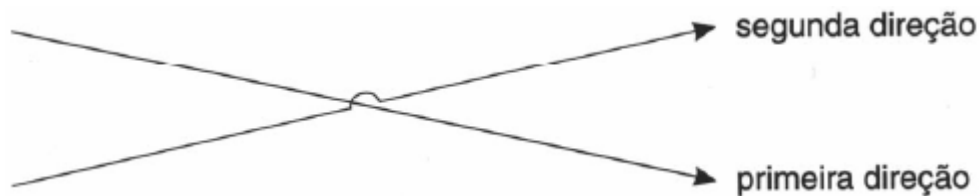
TERMINAL – utilizado na representação de início e fim de um fluxo de um determinado processo, ou programa.	INÍCIO FIM
SETA – indica o sentido do fluxo e conecta os símbolos ou blocos.	
PROCESSAMENTO – Bloco ou símbolo que indica cálculos algoritmos.	
ENTRADA DE DADOS – Apresenta a entrada de dados manual.	
ENTRADA E SAÍDA DE DADOS – representa um dispositivo de entrada ou saída de dados.	
SAÍDA DE DADOS EM VÍDEO – símbolo utilizado para representar a informação ou dados que é visualizado em vídeo.	
SAÍDA DE DADOS EM IMPRESSORA – este símbolo é utilizado quando desejamos apresentar os dados impressos.	
DECISÃO – indicado para apresentação de tomada de decisão.	
CONECTOR – utilizado quando necessário particionar o diagrama.	
CONECTOR – este símbolo indica conexão do fluxo em outra página.	

SÍMBOLOS ESPECIAIS

TECLADO – informações recebidas ou fornecidas.	
DISPLAY – informações exibida através de dispositivo visual, vídeo ou monitor.	
DADOS SEQUÊNCIAIS – memória de massa para armazenamento de dados (fita magnética, streamer).	
CARTÃO PERFURADO – representa cartão de dados ou instruções.	
ARMAZENAMENTO EM DISCO – representa armazenamento de informações.	
DOCUMENTOS MULTIPLOS.	
DOCUMENTO / ARQUIVO.	

3.3 Elaboração do fluxograma

Alguns cuidados necessário durante a elaboração do fluxograma são:

- Quando houver necessidade de cruzamento de linhas, deve ser usado um pequeno arco para esclarecer que as linhas não se tocam, além de indicarem a ordem de ocorrência.

- O sentido de circulação no fluxo é dado pelas linhas de comunicação que fornecem a seqüência das operações e a fluência das informações.
- A comunicação deve seguir a direção natural da leitura, ou seja, de cima para baixo e da esquerda para direita. Tudo o que estiver em sentido inverso deve estar claramente identificado.
- Evite o uso de linhas de comunicação muito longas. Dê preferência ao uso de linhas horizontais e verticais, evitando as diagonais e inclinadas pois elas poluem mais o diagrama do que ajudam.
- O fluxograma só pode ser finalizado com: um arquivo temporário ou definitivo; encerramento do fluxo com o símbolo terminal; destruição do documento final; ou conexão, ligação ou transferência para outro fluxo.
- O excesso de detalhes pode complicar mais do que explicar, portanto esteja atento em encontrar o meio termo.

3.4 Vantagens da utilização de fluxogramas

Podemos apontar as seguintes vantagens:

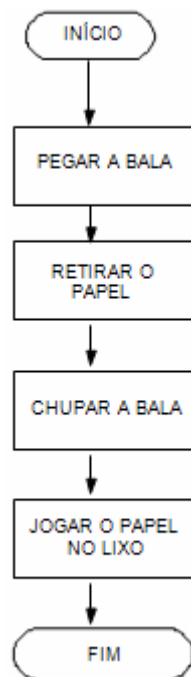
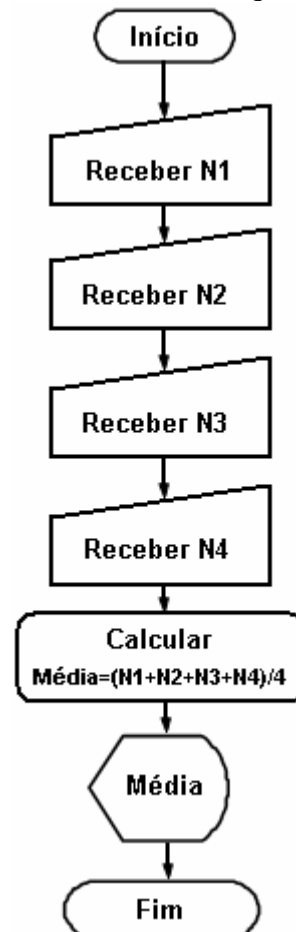
- Descreve qualquer tipo de rotina, desde a mais simples à mais complexa;
- Adequado para descrever relações típicas de empresas de qualquer área.;
- Permite a visão global do universo que está em estudo;
- Descreve como o sistema funciona em todos os componentes envolvidos;
- Restringe a quantidade de interpretações devido à padronização dos símbolos;
- Auxilia na localização de falhas e/ou deficiências descrevendo as repercussões;
- Auxilia na análise de modificações exibindo todos os pontos que serão por elas afetados;
- Facilita a inclusão de atualizações ou modificações, exibindo os pontos de alteração de forma clara e imediata;
- Permite a comparação entre vários fluxos ou várias alternativas de solução de problemas;
- Padroniza as eventuais transições e facilita o trabalho de leitura posterior.

3.5 Desvantagens

As mais importantes que podem ser destacadas não são originárias da ferramenta em si, mas dos profissionais que a utilizam. São elas:

- Vício no uso exclusivo de fluxogramas, não percebendo as implicações técnicas com outras ferramentas;
- É um esquema, um diagrama e portanto, nunca irá detalhar a realidade com o envolvimento de pessoas que fazem o sistema vivo e dinâmico;
- Em nome da simplicidade, acabamos omitindo pequenas informações que muitas vezes são cruciais ao sistema;
- Os símbolos apresentados permitem variações e adaptações, onde o analista cria uma série de aplicações pessoais e particulares que ninguém (somente ele) entende.

3.6 Exemplos de fluxogramas

Dentro do símbolo sempre teremos algo escrito, pois somente os símbolos não nos dizem nada. Veja no exemplo a seguir:

“Chupar uma bala” “Calcular uma média com quatro notas”

Observe que no exemplo da bala seguimos uma seqüência lógica somente com informações diretas, já no segundo exemplo da média utilizamos cálculo e exibimos o resultado do mesmo.

3.7 Exercícios

3.7.1) Construa um diagrama de blocos que :

- Leia a cotação do dólar
- Leia um valor em dólares
- Converta esse valor para Real
- Mostre o resultado

3.7.2) Desenvolva um diagrama que:

- Leia quatro números
- Calcule o quadrado para cada um
- Some todos
- Mostre o resultado

3.7.3) Construa um algoritmo para pagamento de comissão de vendedores de peças, levando-se em consideração que sua comissão será de 5% do total da venda e que você tem os seguintes dados:

- Identificação do vendedor (código)
- Quantidade de peças
- Preço unitário da peça

Após a construção do diagrama de blocos e do algoritmo desenvolvido, faça um teste de mesa.

3.8 Orientação para o desenvolvimento de programas

3.8.1 Diretrizes para a Documentação de Programas

A documentação de um programa complementa o seu código para torná-lo mais compreensível para quem for analisar os detalhes de sua implementação bem como para facilitar o entendimento de seu funcionamento. Pretende-se, com a cobrança de documentação, criar em você o hábito de registrar, ainda que de forma mínima, detalhes complementares importantes relativos à construção dos programas por você produzidos bem como decisões de projeto tomadas por você durante a sua implementação. Na sua vida profissional, dificilmente você irá desenvolver software apenas por você. Além de embutir no código informações complementares para o seu entendimento e de outras pessoas que necessitem entender a forma como o programa em questão foi concebido e construído, é preciso deixar claro, na interface de usuário, em que estado o programa em execução se encontra e o que é esperado do usuário em em cada estado.

A documentação também é importante para você próprio. Depois de um certo tempo não lembramos mais de decisões de projeto tomadas quando da confecção de um determinado software. Uma documentação de um software deve registrar estas decisões para facilitar a manutenção deste software. A documentação, portanto, é tão importante quanto o código. Por esta razão, documente bem aquilo você produz!

A documentação de um programa pode estar embutida no próprio programa e/ou ser fornecido na forma de documentos complementares, como um manual de usuário. Na disciplina, a exigência mínima é um código bem comentado e bem estruturado. É muito importante que você se preocupe com a apresentação do código do programa de modo a facilitar sua leitura. As linhas no código devem ser alinhadas de modo a refletir a estrutura do programa. Comandos internos a outro devem ser deslocados à direita.

Comentários devem ser colocados em pontos estratégicos, presentes em todas as partes onde o código do programa não é trivial ou onde o papel de alguma variável não seja de compreensão imediata. Não comente o óbvio! Além desse tipo de comentários elucidativos, no mínimo os seguintes comentários devem ser inseridos em cada programa produzido por você:

- Informações de caráter administrativo:
  - Nome e RA do autor do programa.
  - A identificação da atividade a que se refere o programa.
  - Data em que foi produzido o programa.
- Descrição informal e sucinta do que o programa faz. Esta descrição deverá ser feita, de maneira sintética, e não deverá transcrever literalmente, comando por comando, em Português.
- Explicações sobre como operar o programa, ressaltando quais são as entradas, como serão solicitadas e quais são as saídas esperadas.
- Breve descrição dos algoritmos utilizados na resolução do problema proposto. Aqui cabe novamente a observação acima. Não transcreva para o Português o seu código em C. Abstraia detalhes de implementação.Redija descrições em nível mais conceitual. Algoritmo e programa são coisas distintas. Um algoritmo serve de base para uma implementação de um software sendo que a partir de um algoritmo podemos derivar diversas implementações alternativas.
- As condições de contorno, ou seja, qual é o conjunto válido de dados de entrada para o correto funcionamento do programa. Em que situação o seu programa não funcionará?

3.8.2 Envio de dúvidas

Um programa ou trecho de código-fonte submetido para avaliação deve estar aderente às "Diretrizes para a Documentação de Programas". O assunto da mensagem deve conter:

[MC102Z] RA000000 – Exercício número

ou

[MC102Z] RA000000 – Dúvida sobre <assunto/aula Nº>

No corpo da mensagem, forneça as seguintes informações:

Descrição dos erros ocorridos e da(s) estratégia(s) utilizada(s) para o teste do programa. A fase de teste é outra etapa importante no processo de desenvolvimento de software. O processo de concepção de um software deve levar em conta a questão sobre como testar o que foi construído. Obviamente não é possível verificar todas as possibilidades de uso de um software. É necessário, contudo, executar uma série adequada de testes que dê um grau mínimo de confiança de que o artefato de software construído está funcionando de forma aceitável. O ideal seria que pessoas distintas das que tivessem elaborado um programa elaborassem as seqüências de teste baseadas apenas na descrição do que o programa faz e não como o programa resolve o problema em questão.

- Quais foram as dificuldades encontradas.
- O que não foi feito.
- O que foi feito a mais.
- Sugestões sobre possíveis extensões e melhorias do programa.
- Comentários pessoais sobre o que julgar relevante.

Caso seja necessário enviar anexos, comunique primeiro através da página de contato da disciplina e aguarde as orientações na mensagem de resposta.

monitoria102@gmail.com

4 - Constantes, Variáveis e Tipos de Dados

Variáveis e constantes são os elementos básicos que um programa manipula. Uma variável é um espaço reservado na memória do computador para armazenar um tipo de dado determinado. Variáveis devem receber nomes para poderem ser referenciadas e modificadas quando necessário. Um programa deve conter declarações que especificam de que tipo são as variáveis que ele utilizará e às vezes um valor inicial. Os tipos primitivos são: inteiros, reais e caracteres. As expressões combinam variáveis e constantes para calcular novos valores.

4.1 Constantes

Constante é um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um programa. Conforme o seu tipo, a constante é classificada como sendo numérica, lógica e literal.

4.2 Variáveis

Variável é a representação simbólica dos elementos de um certo conjunto. Cada variável corresponde a uma posição de memória, cujo conteúdo pode se alterado ao longo do tempo durante a execução de um programa. Embora uma variável possa assumir diferentes valores, ela só pode armazenar um valor a cada instante.

4.3 Tipos de Variáveis

As variáveis e as constantes podem ser basicamente de quatro tipos:

- Numéricas: Específicas para armazenamento de números, que posteriormente serão utilizados para cálculos. Podem ser ainda classificadas como Inteiras ou Reais. As variáveis do tipo inteiro são para armazenamento de números inteiros e as reais são para o armazenamento de números que possuam casas decimais.

- Caracteres: Específicas para armazenamento de caracteres (somente letras).

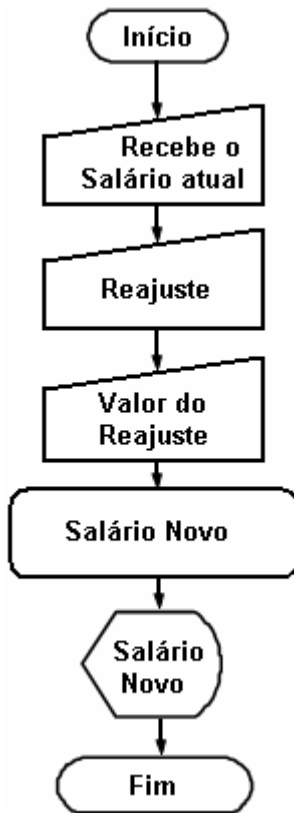
4.4 Exercícios

4.4.1) O que é uma constante? Dê dois exemplos.

4.4.2) O que é uma variável? Dê dois exemplos.

4.4.3) Complete a tabela do teste de mesa abaixo. Faça um diagrama de blocos com os dados da tabela, onde o usuário informa apenas o salário. A seguir, escreva o algoritmo do programa.

Salário	Abono	Total
460,00	239,20	699,20
618,00	321,36	
750,00		
1.270,50		

4.4.4) Considere um programa onde é informado o salário e o reajuste de um funcionário (15%, por exemplo). Observe o fluxograma abaixo. Está correto? (argumente). Reescreva o fluxograma que considera correto e seu respectivo algoritmo.

4.5 Exercícios: Laboratório 2

Descreva os passos utilizados para obter sucesso.

http://www.plastelina.net/games/game1.html

http://www.plastelina.net/games/game2.html

http://www.plastelina.net/games/game3.html

Entregar: descreva somente o resultado final.

http://www.profcardy.com/desafios/aplicativos.php?id=11

Casa 1	Casa 2	Casa 3	Casa 4	Casa 5

5 – Introdução à linguagem C

Nos materiais anteriores, a preocupação foi introduzir o aluno ao meio da programação, apresentando o ambiente onde serão escritos os código-fonte nas aulas de laboratório, assim como a lógica de programação necessária para tal.

Os materiais até então trouxeram noções e até alguns códigos escritos em C, o que será explorado em maior profundidade a partir de agora.

Antes de começar a transformar seus algoritmos em código-fonte, é bom saber que esta linguagem possui algumas características relevantes. Uma delas é o conjunto de palavras reservadas, ou seja, palavras que o programador não deve utilizar para nomear suas variáveis. São elas:

auto	const	double	float	int	short	struct	unsigned
break	continue	Else	for	long	signed	switch	void
case	default	Enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

Um programa em linguagem C, em geral, possui a seguinte estrutura:

```
#include <biblioteca>

declarações globais

<tipo devolvido> main ( lista de parâmetros )
{
      seqüência de comandos;
}

<tipo devolvido> <sua função> ( lista de parâmetros )
{
      seqüência de comandos;
}
```

5.1 Tipos básicos da linguagem C

Há cinco tipos básicos de dados em C: caractere, número inteiro, número real (ponto flutuante), número real (ponto flutuante) de precisão dupla e sem valor (char, int, float, double e void, respectivamente). Observaremos que os demais tipos de dados são derivados de algum destes tipos. O tamanho e a faixa destes tipos de dados variam de acordo com o tipo de processador e com a implementação do compilador C. Um caractere ocupa geralmente 1 byte, enquanto um número inteiro tem normalmente 2 bytes, mas você não pode fazer esta suposição se quiser que seus programas sejam portáveis a uma gama mais ampla de computadores. O padrão ANSI estipula apenas a faixa mínima de cada tipo de dado, não o seu tamanho em bytes.

Números inteiros geralmente correspondem ao tamanho natural de uma palavra do computador em uso, enquanto um número real depende de como eles são implementados. Valores do tipo char são normalmente usados para conter valores definidos pelo conjunto de caracteres ASCII. Os valores fora dessa faixa podem ser manipulados diferentemente entre as implementações de C.

A faixa dos tipos float e double é dada em dígitos de precisão. As grandezas destes tipos depende do método usado para representar os números em ponto flutuante. O padrão ANSI especifica que a faixa mínima de um valor em ponto flutuante é de 1E-37 a 1E+37. O número mínimo de dígitos de precisão é exibido na tabela a seguir.

O tipo void declara explicitamente uma função que não retorna valor algum ou cria ponteiros genéricos. Tais utilizações serão abordadas mais adiante.

Tipo	Tamanho em bits	Formato de Leitura ou Escrita	Intervalo de valores
Char ou signed char	8	%c	-128 a 127
unsigned char	8	%c ou %hhu	0 a 255
int ou signed int	16	%d ou %i	-32.768 a 32.767
unsigned int	16	%u	0 a 65.535
short int	16	%hd ou %hi	-32.768 a 32.767
unsigned short int	16	%hu	0 a 65.535
Long int	32	%ld ou %li	-2.147.483.648 a 2.147.483.647
unsigned long int	32	%lu	0 a 4.294.967.295
Long long int	64	%lld ou %lli	9,223E-15 a 9,223E+15
unsigned long long int	64	%iiu ou %I64u	0 a 18,446E+15
Float	32	%f	3,4E-38 a 3,4E+38
double	64	%lf	1,7E-308 a 1,7E+308
Long double	80	%Lf	3,4E-4932 a 3,4E+4932

Em C, o nome de variáveis, funções e outros elementos definidos pelo usuário são chamados de identificadores. Esses identificadores podem variar de um a diversos caracteres. O primeiro caractere deve ser uma letra ou um sublinhado e os caracteres subseqüentes devem ser letras, números ou sublinhados.

Outro ponto relevante é que lembrar que a linguagem é case sensitive, ou seja, há diferença entre: variável, Variável e VARIÁVEL.

Toda variável possui:

- Nome (rótulo);
- Tipo (domínio);
- Valor (conteúdo);
- Escopo (tempo de vida);

Rótulo	Conteúdo
a	20
b	30
soma	50

Programa

```
// Soma dois numeros inteiros e mostra resultado

#include <stdio.h>

int main()
{
   int a, b, soma=0;

   printf("Digite um numero: ");
   scanf("%d", &a);

   printf("Digite outro numero: ");
   scanf("%d", &b);

   soma=a+b;

   printf("A soma de %d + %d e' %d\n", a, b, soma);

   return(0);
}
```

Visualização no monitor

```
>soma
Digite um numero: 5
Digite outro numero: 2
A soma de 5 + 2 e' 7
>_
```

Outros indicadores de escrita e leitura:

Tipo	Representação		
Char	caractere %c	ASCII %d ou %hhd	ASCII %i ou %hhi
Float	decimal %f	científico %e	decimal/cientifico %g
Double	decimal %lf	científico %le	decimal/científico %lg
Long double	decimal %Lf	científico %Le	decimal/científico %Lg

5.2 Exercícios

5.2.1) Faça um programa que leia caractere, número inteiro e número real e mostre as possíveis saídas conforme cada tipo de entrada.

5.2.2) Transforme os algoritmos elaborados nas aulas anteriores (com entradas e saídas) em programas na linguagem C.

6 – Comandos

Vimos quais são os tipos básicos disponíveis na linguagem C e como podemos ler e mostrar seus conteúdos. Agora será apresentado como definir constantes nos programas, outras possibilidades para escrita e leitura de dados e a linearização de fórmulas.

6.1 Constantes

Assim como variáveis, constantes são usadas são usadas para armazenar números e caracteres. Porém não podemos modificar o conteúdo de uma constante durante a execução do programa. Exemplo:

```
#include <stdio.h>

// definindo constantes
#define PI 3.1415926536
#define MSG "O valor de 2xPI e'" // atribui um texto para MSG

void main(void)
{
   float dois_PI;

   dois_PI=2*PI; // dois_PI=6.2831853072

   printf("%s %5.2f\n", MSG, dois_PI);

}
```

O resultado na tela será: **`O valor de 2xPI e' 6.28`**

6.2 Definindo novos tipos

Podemos usar o comando **typedef** para definir novos tipos de variáveis ou abreviar tipos existentes.

```
#include <stdio.h>

typedef enum {false,true} bool; //o tipo bool só armazena 0/false ou 1/true
typedef unsigned char uchar; // o tipo uchar é o mesmo que unsigned char

int main()
{
   bool v1,v2; // tipo booleano ou variável lógica
   uchar a=10;

   v1 = true;  // o mesmo que atribuir 1 para v1

   v2 = false; // o mesmo que atribuir 0 para v2

   printf("%d %d %d\n", v1,v2,a); // mostra na tela 1 0 10

   return 0;
}
```

6.3 Funções matemáticas e resto da divisão inteira

Um programa consiste de uma seqüência de comandos apresentados na função principal (main). Quando várias tarefas são necessárias, as seqüências de comandos dessas tarefas podem ser agrupadas em funções. Esta forma de organização dos programas evita confusão na depuração do programa e provê uma melhor apresentação do código fonte. Estas funções podem ser chamadas a partir da função principal e devem retornar o resultado da tarefa correspondente, para que as demais tarefas possam ser executadas. A própria função principal deve retornar o valor zero, quando termina com sucesso. Várias funções matemáticas, por exemplo, são disponibilizadas para o programador na linguagem C.

Além de incluir as definições de math.h, as funções matemáticas precisa ser compiladas e incorporadas ao programa executável. Isto é feito com o comando: "**gcc exemplo.c –o exemplo –lm**". Outros exemplos de funções matemáticas são: raiz quadrada "**sqrt(x)**", cosseno "**cos(x)**", arco tangente "**atan(x)**", logaritmo neperiano "**ln(x)**" e arredondamento "**round(x)**". Essas funções podem ser encontradas em qualquer manual da linguagem C ou através do comando **man** no GNU/Linux.

```
#include <math.h>
#include <stdio.h>
#define PI 3.1415926536

int main()
{  double a,b;
   int c,d,e;
   a = 1.0;
   b = exp(a);       // b=2.718282
   a = -4.0;
   a = fabs(a);      // a=4.0
   a = pow(a,3.0);   // a=64.0
   b = log10(100);   // b=2.0
   a = sin(PI/4.0); // a=0.707107
   c = 5;
   d = 3;
   e = c/d; // quociente = dividendo / divisor; c=1 (divisão inteira)
   e = c%d; // resto = dividendo % divisor; c=2 (resto da divisão inteira)
   return(0);
}
```

6.4 Manuseio de Caracteres

Lembrando que o tipo char 'puro', ou seja, sem o uso de vetores, permite a leitura e escrita de apenas um único caractere, podemos converter uma entrada em minúsculo para maiúsculo com o programa abaixo. Maiores detalhes sobre conversões de tipos serão vistos nas próximas aulas.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{  char c;
   printf("Digite uma letra (a-z): ");
   c = getchar();
   if (c >= 'a' && c <= 'z')
      c = (char)((int)'A' + (int)c - (int)'a');
   putchar(c);
   putchar('\n');
   getchar(); // pausa para visualizar resultado
   return 0;
}
```

6.5 Linearização de equações

Os programas de computador trabalham com equações numa única linha, ou seja, é necessário adequar as equações para obter o resultado desejado.

Considere o exemplo:

$$\text{Média} = \frac{\text{Nota 1} + \text{Nota 2} + \text{Nota 3} + \text{Nota 4}}{4}$$

Para calcular a média faremos:

Média = Nota1+Nota2+Nota3+Nota4/4 **Errado!**

Note que se utilizar a fórmula errada, a média informada apenas dividirá a Nota4 por quatro, enquanto o correto é dividir a soma de todas as notas por quatro.
Para obter o resultado desejado, basta agrupar a somatória das notas com parênteses.

Média = (Nota1+Nota2+Nota3+Nota4)/4 **Correto!**

Assim, teremos o seguinte código fonte:

```
#include <stdio.h>

int main()
{
   float n1,n2,n3,n4,media;
   printf("Digite a primeira nota: ");
   scanf("%f", &n1);
   printf("Digite a segunda nota: ");
   scanf("%f", &n2);
   printf("Digite a terceira nota: ");
   scanf("%f", &n3);
   printf("Digite a quarta nota: ");
   scanf("%f", &n4);
   media = (n1+n2+n3+n4)/4;
   printf("Media: %.2f\n", media);
   return 0;
}
```

6.6 Exercícios: Laboratório 3

6.6.1) Faça um programa em linguagem C que solicite ao usuário um número real qualquer e mostre o respectivo valor absoluto.

6.6.2) Faça um programa que pergunte a idade do usuário e divida o valor por quatro. Mostre o quociente e resto da operação.

6.6.3) Dada a fórmula a seguir, escreva sua correspondente linearizada para calcular C e F.

$$C = \frac{(F - 32) \times 5}{9}$$

7 – Operadores

Os operadores são meios pelo qual incrementamos, decrementamos, comparamos e avaliamos dados dentro do computador. Temos três tipos de operadores:

- Operadores Aritméticos
- Operadores Relacionais
- Operadores Lógicos

7.1 Operadores Aritméticos

Os operadores aritméticos são os utilizados para obter resultados numéricos. Além da adição (+), subtração (−), multiplicação (*) e divisão (/), temos incremento (++), decremento (−−) e módulo (%). O módulo retorna o resto de uma divisão. Por exemplo `7%4` retornará `3`.

Antes		Operação	Depois		
M	n		soma	m	n
2	3	soma = m + n	5	2	3
2	3	soma = m + n++	5	2	4
2	3	soma = m + ++n	6	2	4

Hierarquia das operações aritméticas (precedência de operadores):

- parênteses mais internos;
- funções matemáticas(sen,cos,tan);
- multiplicação, divisão, resto, soma e subtração (* / % + -).

Veja os exemplos a seguir, considerando as variáveis: a=10; b=5; c=3; d=7

```
R1 = a+b/c*d = 10+5/3*7 = 10+5/21 = 10+0,238095 = 10,238095

R2 = (a+b)/c*d = (10+5)/3*7 = 15/21 = 0,714286

R3 = ((a+b)/c)*d = ((10+5)/3)*7 = (15/3)*7 = 35

R4 = a+(b/c)*d = 10+(5/3)*7 = 10+(1,666667*7) = 21,666667

R5 = (a+b/c)*d = (10+5/3)*7 = 11,666667*7 = 81,666667

R6 = a+b*c%d = 10+5*3%7 = 10+15%7 = 11
```

O decremento e incremento modificam os valores das variáveis:

```
R7 = a++ = 10

R8 = --a = 10
```

Observe que em R7 o valor da variável ‘a’ foi modificado (a = a + 1), mas se mostrarmos R7 aparecerá 10. Logo a seguir foi feito o inverso (a = a – 1), onde R8 mostrará 10. O que acontece nesses exemplos é o pós incremento, ou seja, R7 recebe o valor de ‘a’ e depois ‘a’ sofre incremento. Já no pré decremento, o valor de ‘a’ é decrementado e depois R8 recebe o valor de ‘a’.

É possível fazer pré e pós tratamento de variáveis (tanto incremento como decremento).

O operador de incremento serve para aumentar, enquanto o de decremento serve para reduzir o valor (conteúdo) de uma variável em uma unidade. Sua aplicação é demonstrada a seguir:

```
#include <stdio.h>

int main()
{
   int a=10,b=0;                 // Mostra:
   printf("A=%d\nB=%d\n",a,b); // A=10 B=0
   b=a*a;
   printf("A=%d\nB=%d\n",a,b); // A=10 B=100
   b=a*a++;
   printf("A=%d\nB=%d\n",a,b); // A=11 B=100
   b=a*++a;
   printf("A=%d\nB=%d\n",a,b); // A=12 B=144
   b=a*a--;
   printf("A=%d\nB=%d\n",a,b); // A=11 B=144
   b=a*--a;
   printf("A=%d\nB=%d\n",a,b); // A=10 B=100
   return 0;
}
```

Podemos ainda utilizar algumas formas contraídas de operações:

Comando	Exemplo	Corresponde a:
`+=`	`a += b;`	`a = a + b;`
`-=`	`a -= b;`	`a = a - b;`
`*=`	`a *= b;`	`a = a * b;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`a %= b;`	`a = a % b;`

7.2 Operadores Relacionais

Sempre retornam valores lógicos (verdadeiro ou falso). Para estabelecer prioridades no que diz respeito a qual operação executar primeiro, utilize os parênteses.

Os operadores relacionais são:

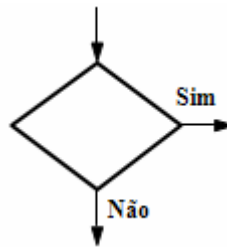
Descrição	Operador
Igual a	`==`
Diferente de	`!=`
Maior que	`>`
Menor que	`<`
Maior ou igual a	`>=`
Menor ou igual a	`<=`

```
#include <stdio.h>

int main()
{
   int verdadeiro, falso;
   verdadeiro = ( 15 < 20);
   falso = ( 15 == 20 );
   printf("Verdadeiro=%d\n", verdadeiro);
   printf("Falso=%d\n", falso);
   return(0);
}

A saída do programa será:
Verdadeiro=1
Falso=0
```

Exemplo:

Considerando duas variáveis inteiras: A=5 e B=3, os resultados das expressões seriam:

Expressão	Resultado
`A == B`	`Falso`
`A != B`	`Verdadeiro`
`A > B`	`Verdadeiro`
`A < B`	`Falso`
`A >= B`	`Verdadeiro`
`A <= B`	`Falso`

Símbolo utilizado para comparações em fluxograma:

7.3 Operadores Lógicos

Os operadores lógicos servem para combinar resultados de expressões, retornando se o resultado final é verdadeiro ou falso.

Os operadores lógicos são:

- E (AND): Uma expressão AND (`&&`) é verdadeira se todas as condições forem verdadeiras.
- OU (OR): Uma expressão OR (`||`) é verdadeira se pelo menos uma condição for verdadeira.
- NÃO (NOT): Um expressão NOT (`!`) inverte o valor lógico da expressão ou condição.

A tabela abaixo mostra todos os valores possíveis criados pelos operadores lógicos:

1º Valor	Operador	2º Valor	Resultado
T	AND	T	T
T	AND	F	F
F	AND	T	F
F	AND	F	F
T	OR	T	T
T	OR	F	T
F	OR	T	T
F	OR	F	F
T	NOT		F
F	NOT		T

```
#include <stdio.h>
// Define o tipo boolean onde
// 0=false e 1=true
typedef enum {false,true} boolean;

int main()
{   boolean v1,v2,v3;
    v1 = true;     // v1=1
    v2 = false;    // v2=0
    v3 = v1&&v2;   // v3=0
    v3 = v1||v2;   // v3=1
    v3 = !v1;      // v3=0
    v3 = ((v1&&v2)||(!v2)); // v3=1
    return(0);
}
```

Exemplo:

Supondo as variáveis: A=5, B=8 e C=1

Expressões			Resultado
A == B	AND	B > C	Falso
A != B	OR	B < C	Verdadeiro
A > B	NOT		Verdadeiro
A < B	AND	B > C	Verdadeiro
A >= B	OR	B = C	Falso
A <= B	NOT		Falso

7.4 Exercícios

7.4.1) Considerando as variáveis SALÁRIO, IR e SALLIQ; e os valores tabelados abaixo, informe se as expressões são verdadeiras ou falsas.

SALÁRIO	IR	SALLIQ	EXPRESSÃO	V ou F
100,00	0,00	100,00	SALLIQ >= 100,00	
200,00	10,00	190,00	SALLIQ < 190,00	
300,00	15,00	285,00	SALLIQ <= SALÁRIO – IR	

7.4.2) Sabendo que A=3, B=7 e C=4, informe se as expressões abaixo são verdadeiras ou falsas.

```
a)[  ]  ( A + C ) > B
b)[  ]  B >= ( A + 2 )
c)[  ]  C == ( B – A )
d)[  ]  ( B + A ) <= C
e)[  ]  ( C + B ) > A
```

7.4.3) Sabendo que A=5, B=4, C=3 e D=6, informe se as expressões abaixo são verdadeiras ou falsas.

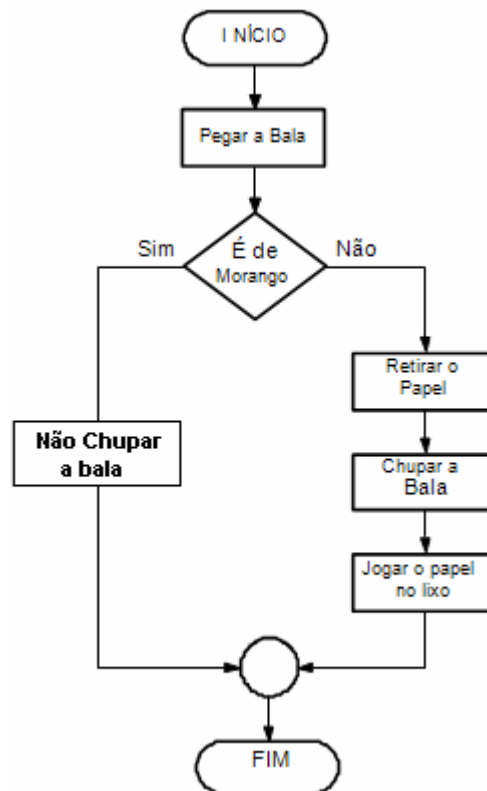
```
a)[  ]  ( A > C ) AND ( C <= D )
b)[  ]  (( A + B ) > 10 ) OR (( A + B ) == ( C + D))
c)[  ]  ( A >= C ) AND ( D >= C )
```

8 – Estruturas de Decisão

Complementando as informações iniciadas com fluxogramas, quando é necessário tomar uma decisão, sempre teremos um resultado VERDADEIRO ou FALSO (símbolo visto no item 7.2).

Assim, a aplicação de uma estrutura de decisão em um algoritmo simples como o "CHUPAR UMA BALA", podemos encontrar situações onde algumas pessoas não gostem de balas de Morango. Neste caso teremos que modificar o algoritmo para:

"Chupar uma bala".

- Pegar a bala
- A bala é de morango?
    - Se sim, não chupe a bala
    - Se não, continue com o algoritmo
- Retirar o papel
- Chupar a bala
- Jogar o papel no lixo

8.1 Exercícios com fluxogramas

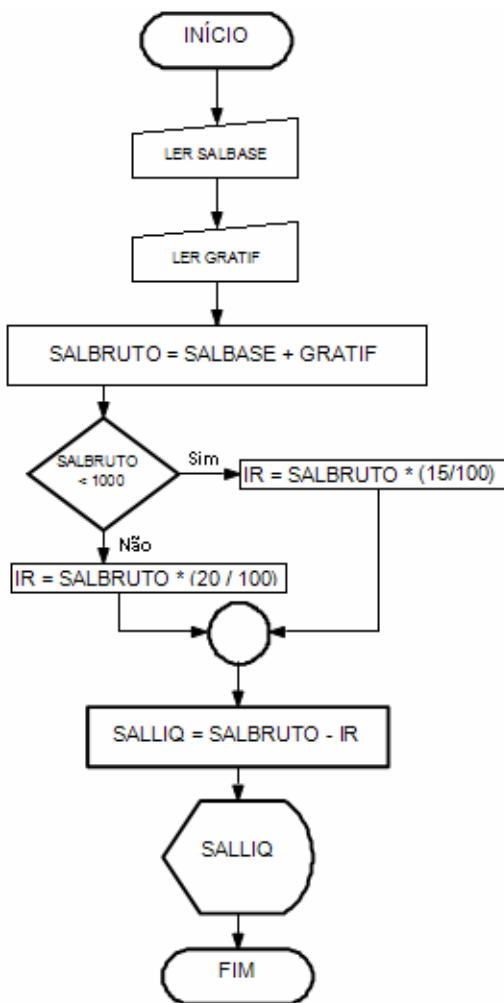
8.1.1) Elabore um diagrama de blocos que leia um número. Se positivo armazene-o em A, se for negativo, em B. No final mostrar o resultado.

8.1.2) Ler um número e verificar se ele é par ou ímpar. Quando for par armazenar esse valor em P e quando for ímpar armazená-lo em I. Exibir P e I no final do processamento.

8.1.3) Construa um diagrama de blocos para ler uma variável numérica N e imprimi-la somente se a mesma for maior que 100, caso contrário imprimi-la com o valor zero.

8.1.4) Tendo como dados de entrada a altura e o sexo de uma pessoa, construa um algoritmo que calcule seu peso ideal, utilizando as seguintes fórmulas:

Para homens: (72.7*h) - 58
Para mulheres: (62.1*h) - 44.7 (h = altura)

8.1.5) Faça um teste de mesa do diagrama apresentado abaixo, de acordo com os dados fornecidos. Elabore um algoritmo baseado no diagrama:

Salbase	Gratif	SalBruto	IR	SalLiq
3.000,00	1.200,00			
1.200,00	400,00			
500,00	100,00			

8.2 Comandos de Decisão

Os comandos de decisão ou desvio fazem parte das técnicas de programação que conduzem a estruturas de programas que não são totalmente seqüenciais. Com as instruções de SALTO ou DESVIO pode-se fazer com que o programa proceda de uma ou outra maneira, de acordo com as decisões lógicas tomadas em função dos dados ou resultados anteriores. As principais estruturas de decisão são: IF..ELSE e SWITCH..CASE.

8.2.1 SE..ENTÃO

A estrutura de decisão SE normalmente vem acompanhada de um comando, ou seja, se determinada condição for satisfeita então execute determinado comando.

```
if (Condição)
   instrução;
```

Equivalente a:

```
if (expressão lógica)
   instrução;
```

Ou SE a condição for satisfeita, execute várias instruções:

```
if (condição)
{
   instrução1;
   instrução2;
   ...
   instruçãoN;
}
```

Imagine um algoritmo que determinado aluno somente estará aprovado se sua média for maior ou igual a 5.0, veja no exemplo de algoritmo como ficaria.

```
if (media >= 5)
   printf("Aluno APROVADO!");
```

Exemplos:

```
// Programa verifica se o numero e’ positivo ou nao
#include <stdio.h>

int main()
{  int n;
   printf(“Digite um número: “);
   scanf(“%d”, &n);
   if (n>=0)
      printf(“Numero positivo”);
   return(0);
}


// Programa recebe dois números, compara-os e mostra resultados
#include <stdio.h>

int main()
{  int a,b;
   printf(“Digite a e b: “);
   /* podemos ler mais de uma variável por linha deixando
      espaço em branco entre os números
   */
   scanf(“%d %d”, &a,&b);

   if ( a > b ) printf(“%d e’ maior que %d\n”,a,b);

   if ( a == b ) printf(“Os numeros sao iguais! %d\n”,a);

   if ( a < b ) printf(“%d e´ menor que %d\n”,a,b);

   if ( a != b ) printf(“%d e´ diferente de %d\n”,a,b);

   return(0);
}
```

8.2.2 SE..ENTÃO..SENÃO

A estrutura de decisão "SE/ENTÃO/SENÃO", funciona exatamente como a estrutura "SE", com apenas uma diferença, em "SE" somente podemos executar comandos caso a condição seja verdadeira, diferente de "SE/SENÃO" pois sempre um comando será executado independente da condição, ou seja, caso a condição seja "verdadeira" o comando da condição será executado, caso contrário o comando da condição "falsa" será executado.

```
if (condicao) /* Se verdadeiro, executa instrucao1 */
   instrucao1;
else          /* Se falso, executa instrucao2 */
   instrucao2;
```

Considerando o exemplo anterior, se média >= 5, aluno aprovado. Caso contrário o aluno deverá fazer exame.

```
if (media >= 5)
   printf("Aluno APROVADO!");
else
   printf("Aluno em EXAME!");
```

Exemplos:

```
// Programa verifica se numero e' par ou impar
#include <stdio.h>
int main()
{  int n;
   printf("Digite um numero: ");
   scanf("%d", &n);
   if (n%2==0)
      printf("Numero par!\n");
   else
      printf("Numero impar!\n");
   return(0);
}
```

Podemos ainda ter estruturas compostas da seguinte maneira:

```
#include <stdio.h>

int main()
{  float n1,n2,media;
   printf("Digite a nota 1: ");
   scanf("%f", &n1);
   printf("Digite a nota 2: ");
   scanf("%f", &n2);
   media=(n1+n2)/2;

   if (media>=7)
      printf("\nMedia: %.2f - Aprovado\n",media);
   else
   {
      if(media>=3)
         printf("\nMedia: %.2f - Exame\n",media);
      else
         printf("\nMedia: %.2f - Reprovado\n",media);
   }
   return(0);
}
```

8.2.3 SELECIONE..CASO

A estrutura de decisão SWITCH..CASE é utilizada para testar, na condição, uma única expressão, que produz um resultado, ou, então, o valor de uma variável, em que está armazenado um determinado conteúdo. Compara-se, então, o resultado obtido no teste com os valores fornecidos em cada cláusula "case".

O conteúdo da variável analisada pode ser um número inteiro ou caractere.

```
SWITCH (variável)
{
   CASE 1: instrução; break;
   CASE 2: instrução; break;
   CASE 3: instrução; break;
   /* caso não seja qualquer anterior, executa a DEFAULT */
   DEFAULT: instrução padrão;
}
```

Exemplos:

```
// Exemplo utilizando numero inteiro
#include <stdio.h>

int main()
{
   int opcao;

   printf("\n1. Saladas");
   printf("\n2. Refeicao");
   printf("\n3. Lanches");
   printf("\n4. Bebidas");
   printf("\n5. Sobremesas");
   printf("\nDigite a opção desejada: ");
   scanf("%d", &opcao);

   switch(opcao)
   {
      case 1: printf("\n\nEscolheu SALADAS\n"); break;
      case 2: printf("\n\nEscolheu REFEICAO\n"); break;
      case 3: printf("\n\nEscolheu LANCHES\n"); break;
      case 4: printf("\n\nEscolheu BEBIDAS\n"); break;
      case 5: printf("\n\nEscolheu SOBREMESAS\n"); break;
      default: printf("\n\nChame o garcom para tirar duvidas\n");
   }
   return(0);
}

// Exemplo com caractere
#include <stdio.h>

int main()
{
   char opcao;

   printf("\nA - Saladas");
   printf("\nB - Refeicao");
   printf("\nC - Lanches");
   printf("\nD - Bebidas");
   printf("\nE - Sobremesas");
   printf("\nDigite a opção desejada: ");
   scanf("%c", &opcao);
```

```
    switch(opcao)
    {  case 'A': printf("\n\nEscolheu SALADAS\n"); break;
       case 'B': printf("\n\nEscolheu REFEICAO\n"); break;
       case 'C': printf("\n\nEscolheu LANCHES\n"); break;
       case 'D': printf("\n\nEscolheu BEBIDAS\n"); break;
       case 'E': printf("\n\nEscolheu SOBREMESAS\n"); break;
       default: printf("\n\nChame o garcom para tirar duvidas\n");
    }
    return(0);
}
```

8.3 Exercícios: Laboratório 4

8.3.1) João comprou um microcomputador para controlar o rendimento diário de seu trabalho. Toda vez que ele traz um peso de peixes maior que o estabelecido pelo regulamento de pesca do estado de São Paulo (50 quilos) deve pagar uma multa de R$ 4,00 por quilo excedente. João precisa que você faça um diagrama de blocos e algoritmo que leia a variável P (peso de peixes) e verifique se há excesso. Se houver, gravar na variável E (Excesso) e na variável M o valor da multa que João deverá pagar. Caso contrário mostrar tais variáveis com o conteúdo ZERO.

8.3.2) Elabore um diagrama de bloco que leia as variáveis C e N, respectivamente código do operário e número de horas trabalhadas. Calcule o salário sabendo-se que ele ganha R$ 10,00 por hora. Quando o número de horas exceder 44 calcule o excesso de pagamento armazenando-o na variável E, caso contrário zerar tal variável. A hora excedente de trabalho vale R$ 20,00. No final do processamento imprimir o salário total pago e o valor do salário excedente.

8.3.3) Faça um diagrama de bloco que leia um número inteiro e mostre uma mensagem indicando se este número é par ou ímpar, e se é positivo ou negativo.

8.3.4) A Secretaria de Meio Ambiente que controla o índice de poluição mantém 3 grupos de indústrias que são altamente poluentes do meio ambiente. O índice de poluição aceitável varia de 0,05 até 0,25. Se o índice sobe para 0,3 as indústrias do 1º grupo são intimadas a suspenderem suas atividades, se o índice crescer para 0,4 as industrias do 1º e 2º grupo são intimadas a suspenderem suas atividades, se o índice atingir 0,5 todos os grupos devem ser notificados a paralisarem suas atividades. Faça um diagrama de bloco que leia o índice de poluição medido e emita a notificação adequada aos diferentes grupos de empresas.

8.3.5) Elabore um algoritmo em leia a idade de um nadador e classifique-o em uma das seguintes categorias:

- Infantil A = 5 a 7 anos
- Infantil B = 8 a 11 anos
- Juvenil A = 12 a 13 anos
- Juvenil B = 14 a 17 anos
- Adultos = Maiores de 18 anos

8.3.6) Construa um algoritmo que leia 500 valores inteiros e positivos e:

- Encontre o maior valor
- Encontre o menor valor
- Calcule a média dos números lidos

8.3.7) Dados três números inteiros informados pelo usuário, apresentá-los em ordem crescente e a seguir, em ordem decrescente.

9 – Estruturas de Repetição

Em muitas tarefas de programação, desejamos que um bloco de comandos seja executado repetidamente até que determinada condição seja satisfeita.

9.1 Comando DO..WHILE

O comando do while é uma instrução de repetição, onde a condição de interrupção é testada após executar o comando.

```
do
{
   bloco de comandos;
} while (expressão lógica);
```

O bloco de comandos é repetido até que a expressão seja falsa. Suponha, por exemplo, um programa para dizer se um número inteiro lido da entrada padrão é par ou ímpar. Desejamos que o programa execute até digitarmos um número negativo.

```
#include <stdio.h>
int main()
{
   int n;
   do {
      printf("Digite um numero inteiro:");
      scanf("%d",&n);
      if ((n%2)==0)
         printf("Numero par\n");
      else
         printf("Numero impar\n");
      } while (n >= 0);
   return(0);
}
```

Um exemplo prático interessante é o algoritmo de Euclides para calcular o Máximo Divisor Comum (MDC) de dois números inteiros positivos.

1. Leia m e n.
2. Faça x ← m e y ← n.
3. Atribua a r o resto da divisão de x por y.
4. Faça x ← y e y ← r.
5. Volte para a linha 3 enquanto r for diferente de zero.
6. Diga que x é o MDC de m e n.

Codifique este algoritmo em C.

9.2 Comando WHILE

O comando while é uma instrução de repetição, onde a expressão lógica é testada antes de executar o comando. Sua estrutura básica envolve quatro etapas:

1. inicialização de uma variável de controle;
2. teste de interrupção envolvendo a variável de controle;
3. execução do bloco de comandos;
4. atualização da variável de controle.

```
inicialização
while(expressão lógica)
{
   bloco de comandos;
   atualização;
}
```

Suponha, por exemplo, um programa que soma n valores reais lidos da entrada padrão e apresenta o resultado na saída padrão.

```
#include <stdio.h>
int main()
{
   int i,n; /* variavel de controle i */
   float soma,num;

   printf("Entre com a quantidade de numeros a serem somados:");
   scanf("%d",&n);

   /* inicializacao */
   i = 1; soma = 0.0;

   while (i <= n) /* expressao */
   {
      /* bloco de comandos */
      printf("Digite o %do. numero:",i);
      scanf("%f",&num);
      soma = soma + num;
      i = i + 1; /* atualizacao */
   }
   printf("O resultado da soma e’ %f\n",soma);
   return(0);
}
```

Modifique o programa acima para calcular o produto dos n números.

Observe que nos exemplos envolvendo o comando do while, a variável de controle é inicializada e atualizada no bloco de comandos. Neste caso, a construção com do while fica mais elegante do que com o comando while. Por outro lado, em situações onde a variável de controle não participa do bloco principal de comandos, a construção fica mais elegante com o comando while.

Podemos combinar vários comandos if, do while e while aninhados.

```
#include <stdio.h>

int main()
{
   int i,n;
   float soma,num;
   char opt;

   do
   {
      printf("\nDigite s para somar numeros, \n");
      printf("ou qualquer outra letra para sair do programa.\n");
      printf("Digite a opcao desejada:\n");
      scanf(" %c",&opt);

      if (opt == 's')
      {
```

```
          printf("\n..:: Soma ::..\n");
          printf("Entre com a quantidade de números: \n");
          scanf("%d",&n);
          i = 1; soma = 0.0;
          while (i <= n)
          {
             printf("Digite o %do. numero:",i);
             scanf("%f",&num);
             soma = soma + num;
             i = i + 1;
          }
          printf("O resultado da soma e %f\n",soma);
       }
    } while (opt == 's');
    return(0);
 }
```

Podemos também criar um laço infinito, substituindo do while por while(1), e sair dele com o comando break no else do if. Outro comando de desvio interessante é o comando continue. Ele permite desconsiderar o restante do bloco de comandos, voltando para o teste da expressão lógica. Por exemplo, podemos desprezar o processamento de números negativos, acrescentando:

```
if (num < 0) continue;
   após a leitura do número.
```

9.3 Comando FOR

O comando for é uma simplificação do comando while, onde a inicialização da variável de controle, a expressão lógica envolvendo a variável de controle e a atualização da variável são especificadas no próprio comando. Sua implementação é feita com o comando while, portanto seu comportamento é o mesmo: após a inicialização, a expressão lógica é testada. Se for verdadeira, o bloco de comandos é executado. Após execução, a variável de controle é atualizada, a expressão lógica é verificada, e o processo se repete até que a expressão seja falsa.

```
for (inicialização; expressão; atualização)
{
      bloco de comandos
}
```

Por exemplo, um programa para somar n números fica.

```
#include<stdio.h>
int main()
{
   int n,i;
   float soma,num;
   printf("\n..:: Soma ::..\n");
   printf("Entre com a quantidade de numeros: ");
   scanf("%d",&n);
   for (i=1, soma = 0.0; i <= n; i=i+1, soma = soma + num)
   {
      printf("Digite o %do. numero:",i);
      scanf("%f",&num);
   }
   printf("O resultado da soma eh %f\n",soma);
   return(0);
}
```

```
// Programa que mostra os numeros pares de 0 a 10
#include <stdio.h>

int main()
{
   int a;
   for(a=0;a<=10;a+=2)
      printf("%d ",a);  // Mostra: 0 2 4 6 8 10
   return 0;
}
```

Outro exemplo é um programa para calcular o maior entre n números lidos da entrada padrão.

```
#include <stdio.h>
int main()
{
   int i,n,num,maior;
   printf("Entre com a quantidade de numeros: ");
   scanf("%d",&n);
   maior = 0;
   for(i=1; i <= n; i++)
   {
      printf("Entre com um inteiro: ");
      scanf("%d",&num);
      if (num > maior)
         maior = num;
   }
   printf("O maior inteiro lido foi %d\n",maior);
   return(0);
}
```

Sabendo que o fatorial de um número inteiro n é n X (n - 1) X (n - 2) . . . 1, faça um programa para calcular o fatorial de um número lido da entrada padrão usando o comando for e apenas duas variáveis.

O triângulo de Floyd é formado por n linhas de números consecutivos, onde cada linha contém um número a mais que a linha anterior. Para imprimir o triângulo de Floyd precisamos aninhar um for dentro do outro.

```
#include <stdio.h>
int main()
{
   int l,c,nl,i;
   printf("Entre com o numero de linhas: ");
   scanf("%d",&nl);
   i = 1;
   for(l=1; l <= nl; l++)
   {
      for(c=1; c <= l; c++)
      {
         printf("%2d ",i);
         i++;
      }
      printf("\n");
   }
   return(0);
}
```

Outro exemplo que requer dois comandos for aninhados é a impressão de uma tabuada. Faça um programa para imprimir uma tabuada com n linhas e n colunas.

Observe que existe uma relação entre a integral de uma função contínua, sua aproximação pela somatória de uma função discreta, e a implementação da somatória usando o comando for. Por exemplo,

$$\int_{x_{\min}}^{x_{\max}} (ax+b)d_x \approx \sum_{x=x_{\min}+d_x/2}^{x=x_{\max}-d_x/2} (ax+b)d_x,$$

onde $x_{min} < x_{max}$, é uma aproximação para a integral da curva, a qual tem maior exatidão para valores menores de $d_x > 0$. Podemos implementar esta integral como:

```
#include<stdio.h>
int main()
{
   float a,b,xmin,xmax,x,dx,integral;
   printf("Entre com os coeficientes a e b da reta y=ax+b: \n");
   scanf("%f %f",&a,&b);
   printf("Intervalo [xmin,xmax] para calculo de area: \n");
   scanf("%f %f",&xmin,&xmax);
   printf("Entre com o incremento dx: \n");
   scanf("%f",&dx);
   integral = 0.0;
   for (x=xmin+dx/2.0; x <= xmax-dx/2.0; x=x+dx)
      integral += (a*x+b)*dx;
   printf("integral %f\n",integral);
   return(0);
}
```

9.4 Exercícios

9.4.1) Faça um algoritmo que determine o maior entre N números. A condição de parada é a entrada de um valor 0, ou seja, o algoritmo deve ficar calculando o maior até que a entrada seja igual a 0 (ZERO).

9.4.2) Uma rainha requisitou os serviços de um monge e disse-lhe que pagaria qualquer preço. O monge, necessitando de alimentos, indagou à rainha sobre o pagamento, se poderia ser feito com grãos de trigo dispostos em um tabuleiro de xadrez, de tal forma que o primeiro quadro deveria conter apenas um grão e os quadros subseqüentes, o dobro do quadro anterior. A rainha achou o trabalho barato e pediu que o serviço fosse executado, sem se dar conta de que seria impossível efetuar o pagamento. Faça um algoritmo para calcular o número de grãos que o monge esperava receber.

9.4.3) Faça um algoritmo que conte de 1 a 100 e a cada múltiplo de 10 emita uma mensagem: "Múltiplo de 10".

9.4.4) Escreva um programa cuja saída em vídeo possua m linhas com i asteriscos alinhados a esquerda, onde i corresponde ao número da linha corrente.
Exemplo: Para m=4, a saída do programa deve ser:

```
*
**
***
****
```

10 – Vetores

Nas aulas anteriores vimos que uma variável simples está associada a uma posição de memória e qualquer referência a ela significa um acesso ao conteúdo de um pedaço de memória, cujo tamanho depende de seu tipo. Veremos agora mais um dos tipos simples de estrutura de dados, denominada vetor.

O vetor permite associar um identificador a um conjunto de variáveis simples de um mesmo tipo, cuja sintaxe apropriada será apresentada a seguir.

```
int main()
{
   tipo identificador[quantidade de variáveis];
}
```

Um vetor é um conjunto de posições consecutivas de memória, identificadas por um mesmo nome, individualizadas por índices e cujo conteúdo é do mesmo tipo.

Consideremos a leitura de 10 notas de alunos e após o cálculo da média geral da classe, verificar e mostrar as notas acima da média. Precisamos armazenar cada nota em uma variável para realizar tal operação e assim, utilizando um vetor podemos ter o conjunto de 10 notas associado a um único identificador. Imagine a situação mencionada para dezenas de alunos. A criação de uma variável para a nota de cada aluno se tornaria algo complicado de programar. Com um vetor para as 10 notas, teremos:

```
      float nota[10];
```

A referência ao conteúdo da n-ésima variável é indicada pela notação nota[n-1], onde n é uma expressão inteira ou uma variável inteira. Em outras palavras, o índice do vetor será de zero (0) a (quantidade-1).

```
/* Programa: Melhores notas */
#include <stdio.h>

int main()
{
   float nota[10],media=0.0;
   int i;

   printf("Entre com as notas dos alunos\n");

   for (i=0; i<10; i++)
   {
      printf("Digite a nota %2d: ",i+1);
      scanf("%f", &nota[i]);
      media += nota[i];
   }
   media /= 10.0;
   printf("\nMedia da classe: %.2f\n\n",media);

   for (i=0; i<10; i++)
   {
      if (nota[i] > media)
         printf("Nota[%d]=%.2f\n",i+1,nota[i]);
   }

   return 0;
}
```

2.5	4.5	9.0	7.0	5.5	8.3	8.7	9.3	10.0	9.0
0	1	2	3	4	5	6	7	8	9

Figura 1: Vetor com dez notas representado pelo identificador nota

```
Administrador@Windows ~
$ ./vetor
Entre com as notas dos alunos
Digite a nota  1: 2.5
Digite a nota  2: 4.5
Digite a nota  3: 9.0
Digite a nota  4: 7.0
Digite a nota  5: 5.5
Digite a nota  6: 8.3
Digite a nota  7: 8.7
Digite a nota  8: 9.3
Digite a nota  9: 10
Digite a nota 10: 9

Media da classe: 7.38

Nota[3]=9.00
Nota[6]=8.30
Nota[7]=8.70
Nota[8]=9.30
Nota[9]=10.00
Nota[10]=9.00

Administrador@Windows ~
$
```

Figura 2: Saída do programa melhores notas

O identificador é uma variável do tipo apontador, cujo conteúdo é o endereço de memória do primeiro elemento do vetor. Considere, por exemplo, o trecho de código abaixo:

```
int main()
{
   int v[5]={7,2,1,4,10}; // inicializacao de um vetor com 5 posicoes do tipo
inteiro

   printf(“Endereço de v[0]: %d=%d\n”,&v[0],v);
   printf(“Conteudo de v[0]: %d=%d\n”,v[0],*v);
   printf(“Conteudo de v[1]: %d=%d\n”,v[1],*(v+1));
   printf(“Conteudo de v[4]: %d=%d\n”,v[4],*(v+4));
   printf(“Endereço de v[4]: %d=%d\n”,&v[4],(v+4));
   return 0;
}
```

Na memória, os elementos do vetor são armazenados um após o outro. O endereço de v[i] pode ser obtido pro &v[i] ou (v+i), já que v guarda o endereço de v[0] e, portanto, (v+i) guarda o endereço de v[i]. Ao somarmos (v+i), pulamos na memória i*sizeof(int) bytes a partir do endereço de v[0].

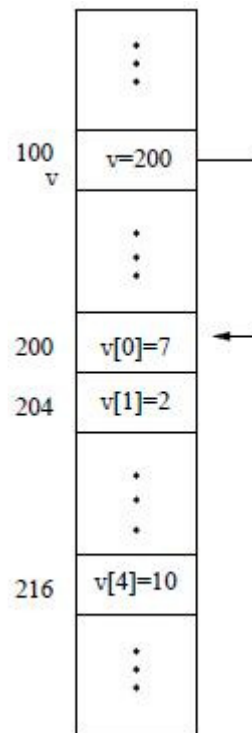



Figura 3: Como o vetor fica armazenado na memória. Os números ao lado indicam o endereço de memória na pilha. As variáveis e os respectivos conteúdos são indicados nos escaninhos.

10.1 Busca em vetores

Um problema comum quando se manipula vetores é a necessidade de encontrar um elemento com um dado valor. Uma forma trivial de fazer este acesso é percorrer do índice inicial ao final todos os elementos do vetor até achar o elemento desejado. Esta forma de busca é chamada linear, pois no pior caso o número de comparações necessárias é igual ao número de elementos no vetor.

10.1.1 Busca Linear

Suponha, por exemplo, que desejamos saber se existe uma nota x no vetor lido.

```
#include <stdio.h>

int main()
{
   float nota[11], x; /* vetor criado com uma posição a mais */
   int i;

   printf(“Entre com 10 notas\n”);
   for (i=0; i<10; i++)
      scanf(“%f”, &nota[i]);

   while(1)
   {
      printf(“Digite a nota procurada ou -1 para sair do programa\n”);
      scanf(“%f”, &x);

      if (x==-1.0)
         break;
      /* busca linear */
      nota[10]=x; /* elemento sentinela */
```

```
      i=0;
      while(nota[i] != x) /* busca com sentinela */
         i++;

      if (i <10)
         printf("nota %.2f encontrada na posição %d\n", nota[i], i);
      else
         printf("nota %.2f não encontrada\n", x);
   }
   return 0;
}
```

Imagine agora que nosso vetor tenha o tamanho 1024. O que podemos fazer para reduzir o número de comparações? Quanto maior for a quantidade de informação sobre os dados, mais vantagens podemos tirar para agilizar os algoritmos.

10.1.2 Busca Binária

A busca binária, por exemplo, reduz o número de comparações de n para $\log_2(n)$ no pior caso, onde n é o tamanho do vetor. Um vetor de tamanho $1024 = 2^{10}$ requer no pior caso 10 comparações. No entanto, a busca binária requer que o vetor esteja ordenado. Esta ordenação também tem um custo a ser considerado, mas se vamos fazer várias buscas, este custo pode valer a pena. A idéia básica é que a cada iteração do algoritmo, podemos eliminar a metade dos elementos no processo de busca. Vamos supor, por exemplo, que o vetor de notas está em ordem crescente.

```
#include <stdio.h>

typedef enum {false,true} bool;

int main()
{
   float nota[10],x;
   int i,pos,inicio,fim;
   bool achou;

   printf("Entre com 10 notas em ordem crescente\n");
   for (i=0; i<10; i++)
      scanf("%f", &nota[i]);

   while(1)
   {
      printf("Digite a nota procurada ou -1 para sair do programa\n");
      scanf("%f", &x);

      if (x==-1.0)
         break;

      /* busca binária */
      inicio=0;
      fim=9;
      achou=false;

      while((inicio<=fim)&&(!achou))
      {
         pos=(inicio+fim)/2;
         if(x < nota[pos])
            fim=pos-1;
         else
            if(x > nota[pos])
               inicio=pos+1;
```

```
            else
               achou=true;
      }
      if(achou)
         printf("nota %.2f encontrada na posição %d\n", nota[pos],pos);
      else
         printf("nota %.2f nao encontrada\n",x);
   }
   return 0;
}
```

10.2 Exercícios: Laboratório 5

10.2.1) Faça um programa que permita a um usuário digitar e armazenar dois vetores com 10 números inteiros em cada. Depois de digitado todos os valores, calcular a soma dos elementos de mesma posição dos vetores (elemento a elemento) armazenando em um terceiro vetor. Exibir no monitor os elementos do terceiro vetor.

10.2.2) Faça um programa que leia 20 números reais e permita que o usuário decida qual operação realizar: soma, subtração, multiplicação ou divisão. O vetor resultante de ser R = A <operação> B, onde os valores lidos para o segundo vetor devem ser diferentes de zero.

10.2.3) Com base no exercício 10.2.2, faça um programa, onde dado um valor digitado pelo usuário, pesquise no terceiro vetor (R), utilizando busca linear.

10.2.4) Com base no exercício 10.2.2, faça um programa, onde dado um valor digitado pelo usuário, pesquise no terceiro vetor (R), utilizando busca binária.

10.2.5) Considerando o exemplo para criar números aleatórios entre 0 e 50, faça um programa que preencha um vetor de 500 posições com números inteiros não sinalizados (unsigned int). O programa deve retornar: a) o maior elemento do vetor; b) a média dos valores; c) a posição do menor elemento do vetor; d) os números primos presentes no vetor.

```
#include <stdio.h>
#include <stdlib.h>
main()
{  int i;
   /* inicializar o gerador de números aleatórios com time(NULL) */
   srand(time(NULL));
   for (i=0; i<15; i++)
      printf("%d ", rand() % 50);
   return 0;
}
```

11 – Ordenação

Em muitas situações, como na busca binária, desejamos vetores ordenados para tal. Veremos aqui, ordenação por: seleção, inserção e permutação. Para demonstrar, ordenaremos de forma crescente 10 notas utilizadas no exemplo dado em aula.

11.1 Ordenação por seleção

O algoritmo mais intuitivo é o de ordenação por seleção (selection sort). A idéia básica é percorrer o vetor várias vezes, selecionando o maior elemento e trocando-o com o da última posição ainda não usada.

```
#include <stdio.h>

int main()
{
   float nota[10];
   int i,j,jm;

   printf("Entre com 10 notas: ");
   for (i=0; i<10; i++)
      scanf("%f", &nota[i]);

   /* Colocar em ordem crescente por Selecao */
   for(j=9; j>=1; j--)
   {/* Seleciona a maior nota entre j notas e troca-a com a da posicao j */
      jm=j;
      for(i=0; i<j; i++)
      {
         if(nota[i] > nota[jm])
           jm=i;
      }
      /* troca */
      if (j != jm)
      {  nota[j]  = nota[j] + nota[jm];
         nota[jm] = nota[j] - nota[jm];
         nota[j]  = nota[j] - nota[jm];
      }
   }
   /* Imprime as notas ordenadas */
   for(i=0; i < 10; i++)
      printf("%.2f ",nota[i]);

   printf("\n");
   return 0;
}
```

Observe que no pior caso teremos (n(n-1))/2 comparações entre notas e (n-1) trocas, onde n é o tamanho do vetor. Dizemos então que o algoritmo executa em tempo proporcional ao quadrado do número de elementos e adotamos a notação $O(n^2)$. O processo de ordenação também não usa nenhum vetor auxiliar, portanto dizemos que a ordenação é in place. Esta observação se aplica aos outros métodos abaixo.

11.2 Ordenação por inserção

A ordenação por inserção (insertion sort) se assemelha ao processo usado para ordenar cartas de um baralho. A cada passo nós ordenamos parte da seqüência e a idéia para o passo seguinte é inserir a próxima nota na posição correta da seqüência já ordenada no passo anterior.

```
#include <stdio.h>

int main()
{
   float nota[10];
   int i,j;

   printf("Entre com 10 notas: ");
   for (i=0; i<10; i++)
      scanf("%f", &nota[i]);

   /* Colocar em ordem crescente por Insercao */
   for(j=1; j<10; j++)
   {/* Insere a nota da posicao j na posicao correta
       da sequencia ja ordenada da posicao 0 ate j-1 */
      i=j;
      while((i>0)&&(nota[i] < nota[i-1]))
      {  /* troca as notas das posicoes i e i-1 */
         nota[i]   = nota[i] + nota[i-1];
         nota[i-1] = nota[i] - nota[i-1];
         nota[i]   = nota[i] - nota[i-1];
         i--;
      }
   }
   /* Imprime as notas ordenadas */
   for(i=0; i < 10; i++)
      printf("%.2f ",nota[i]);

   printf("\n");
   return 0;
}
```

Observe que no pior caso, o número de comparações e trocas é (n(n-1))/2, onde n é o tamanho do vetor. Portanto, o algoritmo é também $O(n^2)$, apesar do número maior de trocas.

11.3 Ordenação por permutação

A idéia básica é simular o processo de ebolição de uma bolha de ar dentro d´água (buble sort). A cada passo, a maior nota, que representa a bolha, deve subir até a superfície. Ou seja, trocas são executadas do início para o final do vetor, até que a maior nota fique na última posição ainda não usada.

```
#include <stdio.h>

int main()
{
   float nota[10];
   int i,j;
```

```
   printf("Entre com 10 notas: ");
   for (i=0; i<10; i++)
      scanf("%f", &nota[i]);

   /* Colocar em ordem crescente por Permutacao */
   for(j=9; j>0; j--)
   {/* Seleciona a maior nota entre cada par de notas
       consecutivas, faz a troca se necessario, ateh
       que a maior nota seja inserida na posicao j+1 */
      for (i=0; i<j; i++)
      {  if(nota[i] > nota[i+1])
       {  /* troca as notas */
          nota[i]   = nota[i+1] + nota[i];
          nota[i+1] = nota[i]   - nota[i+1];
          nota[i]   = nota[i]   - nota[i+1];
       }
      }
   }
   /* Imprime as notas ordenadas */
   for(i=0; i < 10; i++)
      printf("%.2f ",nota[i]);

   printf("\n");
   return 0;
}
```

No pior caso, temos (n(n-1))/2 comparações e trocas, onde n é o tamanho do vetor. Portanto, o algoritmo é $O(n^2)$ com número de trocas equivalente ao da ordenação por inserção.

11.4 Exercícios

11.4.1) Refaça o programa de busca binária (10.2.4) utilizando cada um dos três algoritmos acima para ordenar o vetor.

11.4.2) Faça um programa que leia dois vetores, com 20 posições de números reais distintos em ordem crescente e gere um terceiro vetor mantendo a ordem por intercalação de valores dos outros dois.

12 – Vetores Multidimensionais (Matrizes)

Os vetores também podem possuir múltiplas dimensões, quando declaramos um identificador que é um vetor de vetores de vetores . . .

```
#define N1 10
#define N2 10
...
#define Nn 50

int main()
{
   tipo identificador[N1][N2]…[Nn];
}
```

No caso bidimensional, por exemplo, o identificador é chamado de matriz e corresponde ao que entendemos no ensino básico por matriz (figura 1), onde são respeitados os índices, conforme visto com vetores.

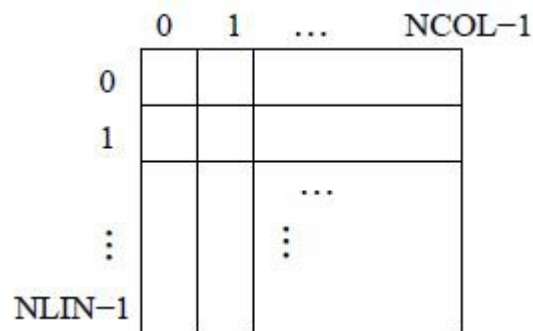



Figura 1: Matriz m[Número de Linhas][Número de Colunas] de inteiros

```
#define NL 80
#define NC 100

int main()
{
    int m[NL][NC];
}
```

Matrizes podem ser utilizadas para cálculos envolvendo álgebra linear, para armazenar imagens, e muitas outras aplicações. O programa abaixo, por exemplo, soma duas matrizes e apresenta a matriz resultante na tela.

```
#include <stdio.h>

#define N 20

int main()
{
    int m1[N][N],m2[N][N],m3[N][N];
    int l,c,nlin,ncol;

    printf("Entre com os numeros de linhas e colunas das matrizes: ");
```

```
    scanf("%d %d", &nlin, &ncol); // considerando nlin e ncol <= 20

    // Leitura da matriz 1
    printf("Entre com os elementos da matriz 1\n");
    for(l=0; l<nlin; l++)
       for(c=0; c<ncol; c++)
          scanf("%d", &m1[l][c]);

    // Leitura da matriz 2
    printf("Entre com os elementos da matriz 2\n");
    for(l=0; l<nlin; l++)
       for(c=0; c<ncol; c++)
          scanf("%d", &m2[l][c]);

    // Soma as matrizes
    for(l=0; l<nlin; l++)
       for(c=0; c<ncol; c++)
          m3[l][c]=m1[l][c]+m2[l][c];

    // Mostra o resultado
    printf("Resultado:\n");
    for(l=0; l<nlin; l++)
    {
       for(c=0; c<ncol; c++)
          printf("%2d ", m3[l][c]);
       printf("\n");
    }
    return 0;
}
```

Outro exemplo é a multiplicação de matrizes. O número de linhas da matriz 2 deve ser igual ao número de colunas da matriz 1.

```
#include <stdio.h>

#define N 20

int main()
{
    int m1[N][N],m2[N][N],m3[N][N];
    int l,c,i,nlin1,ncol1,nlin2,ncol2,nlin3,ncol3;

    printf("Entre com os numeros de linhas e colunas da matriz 1: ");
    scanf("%d %d", &nlin1, &ncol1); // considerando nlin e ncol <= 20

    printf("Entre com os numeros de linhas e colunas da matriz 2: ");
    scanf("%d %d", &nlin2, &ncol2); // considerando nlin e ncol <= 20


    if (ncol1 != nlin2)
    {
       printf("ERRO: Numero de colunas da matriz 1 e' diferente\n");
       printf("      do numero de linhas da matriz 2.\n");
       exit(-1);
    }

    // Leitura da matriz 1
    printf("Entre com os elementos da matriz 1\n");
    for(l=0; l<nlin1; l++)
       for(c=0; c<ncol1; c++)
```

```
            scanf("%d", &m1[l][c]);

    // Leitura da matriz 2
    printf("Entre com os elementos da matriz 2\n");
    for(l=0; l<nlin2; l++)
        for(c=0; c<ncol2; c++)
            scanf("%d", &m2[l][c]);

    nlin3=nlin1;
    ncol3=ncol2;

    // Multiplica as matrizes
    for(l=0; l<nlin3; l++)
        for(c=0; c<ncol3; c++)
        {   m3[l][c]=0;
            for(i=0; i<nlin2; i++)
                m3[l][c]=m3[l][c]+m1[l][i]*m2[i][c];
        }

    // Mostra o resultado
    printf("Resultado:\n");
    for(l=0; l<nlin3; l++)
    {
        for(c=0; c<ncol3; c++)
            printf("%2d ", m3[l][c]);
        printf("\n");
    }

    return 0;
}
```

12.1 Linearização de Matrizes

Matrizes também podem ser representadas na forma unidimensional (isto é muito comum em processamento de imagens, por exemplo). Considere a matriz da figura 1. Podemos armazenar seus elementos da esquerda para a direita e de cima para baixo, iniciando em [0,0] até [NLin-1, NCol-1] em um vetor de NLin X NCol variáveis. Para saber o índice i do elemento do vetor correspondente a variável m[l,c] da matriz, faremos i=l*NCol+c. O processo inverso é dado por c=i%NCol e l=i/NCol. A figura 2 ilustra a linearização de uma matriz m[3][2] em um vetor v[6].

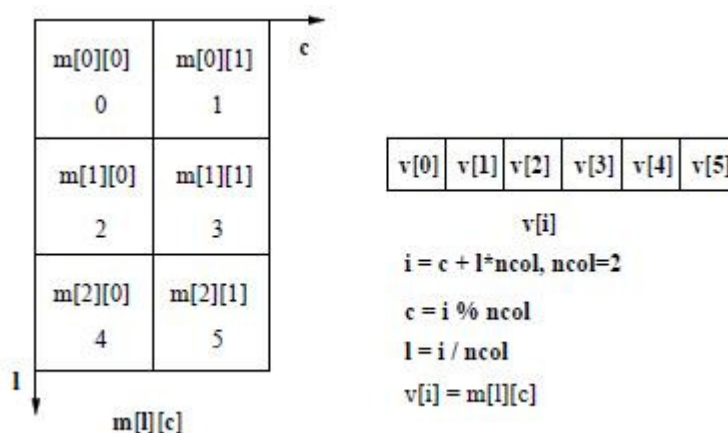



Figura 2: Matriz m[3][2] linearizada em vetor v[6], onde v[i]=m[l][c].

12.2 Exercícios: Laboratório 6

Consulte os livros de álgebra linear e:

12.2.1) Escreva um programa para calcular a transposta de uma matriz.

12.2.2) Escreva um programa para calcular o determinante de uma matriz.

12.2.3) Escreva um programa para inverter uma matriz.

13 – Cadeia de Caracteres (Strings) e Conversões

Uma cadeia de caracteres (string) é uma seqüência de letras, símbolos (!,?,+.+,/,%,&, entre outros), espaços em branco e/ou dígitos terminada por '\0', ou seja, um vetor de variáveis do tipo char.

```
#include <stdio.h>

int main()
{
    char texto[100]="Est? e' uma frase qualquer...";
    int i;

    texto[3]='a'; // Corrige o erro no texto

    i=0;
    while(texto[i] != '\0')
    { // Imprime caractere por caractere separado por |
       printf("%c|", texto[i]);
       i++;
    }
    printf("\n%s\n", texto); // Imprime o texto todo

    return 0;
}
```

13.1 Lendo da entrada padrão

Até o momento vimos que o comando **scanf** pode ser usado para ler dados da entrada padrão (teclado). Mais adiante veremos que **fscanf** possui sintaxe similar, porém é mais genérico. Pois permite a leitura de dados não só da entrada padrão, identificada por **stdin**, como a leitura de dados armazenados em um **arquivo texto** no formato ASCII. A mesma observação se aplica aos comandos **printf** e **fprintf**, com identificador **stdout** para a saída padrão.

```
#include <stdio.h>
int main()
{
    char texto[100];
    /* Le cadeias de caracteres ate encontrar espaco em branco,
       nova linha ou EOF (fim de arquivo).
       Equivalente a scanf("%s",texto);
       O caractere '\0' e' inserido no final do vetor apos
       a leitura */
    fscanf(stdin,"%s",texto);
    printf("\n%s\n",texto);
    return 0;
}
```

A mesma relação é válida entre outros comandos de entrada, tais como **gets**, **fgets**, **getc** (ou **getchar**) e **fgetc**, porém o comando **fgets** é mais seguro do que **gets**, pois especifica o número máximo de caracteres que o vetor suporta.

```
#include <stdio.h>

int main()
{
    char texto[100];
```

```
    /* Le no maximo 99 caracteres, incluindo espacos em branco,
       ate encontrar nova linha ou EOF. Mais seguro que gets(texto);
       O caractere '\0' e' inserido no final do vetor texto,
       apos a leitura */
    fgets(texto,99,stdin);
    printf("\n%s\n",texto);
    return 0;
}
```

Exemplo de um programa que lê caracteres de um arquivo de entrada até preencher o vetor texto ou até o final do arquivo.

```
#include <stdio.h>

int main()
{
    char texto[100];
    int i,c;

    /* Le no maximo 99 caracteres ou ate fim de arquivo
       especificado na entrada padrao. Exemplo de uso:
       programa < arquivo
    */
    i=0;
    while(((c=fgetc(stdin))!=EOF)&&(i<100))
    {   texto[i]=(char)c;
        i++;
    }
    printf("\n%s\n",texto);
    return 0;
}
```

Uma variação deste programa seria ler apenas a primeira linha, caso usássemos o '\n' no lugar de EOF.

13.2 Convertendo cadeias em números e vice-versa

Os comandos sprintf e sscanf funcionam de forma similar aos comandos printf e scanf, porem utilizando uma cadeia de caracteres no lugar da saída/entrada padrão, respectivamente.

```
#include <stdio.h>

int main()
{
    char v1[10],v2[10],texto[100];
    float v3;
    int v4;

    // Converte float para string
    sprintf(v1,"%2.1f",3.4);
    printf("%s\n",v1);

    // Converte int para string
    sprintf(v2,"%2d",12);
    printf("%s\n",v2);

    // Converte string para float
    sscanf(v1,"%f",&v3);
    printf("%2.1f\n", v3);
```

```
    // Converte string para int
    sscanf(v2,"%d",&v4);
    printf("%d\n",v4);

    // Grava numeros em string
    sprintf(v1,"%2.1f %2d",3.4,12);
    printf("%s\n",v1);

    /* Ler numeros de strings. Lembre-se do bug que requer
       espaco em branco extra */
    sscanf(v1," %f %d", &v3, &v4);
    printf("%2.1f %d\n",v3,v4);

    // Gera string com valores formatados
    sprintf(texto,"Os valores sao %2.1f e %d\n",v3,v4);
    printf("%s",texto);

    return 0;
}
```

Em algumas situações desejamos que o programa leia uma seqüência de n valores da entrada padrão. Esses valores podem ser passados, um por linha ou separados por espaços em branco (ou vírgulas).

```
#include <stdio.h>
#define N 50

int main()
{
    float v[N];
    int i,n;

    // Recebe quantidade de numeros reais que serao lidos
    fscanf(stdin,"%d",&n);
    // Le os numeros
    for (i=0;i<n;i++)
       fscanf(stdin,"%f",&v[i]);
    // Mostra numeros lidos
    for(i=0;i<n;i++)
       fprintf(stdout,"%f\n",v[i]);

    return 0;
}
```

Porém, em alguns casos, veremos que esses valores são passados em um cadeia de caracteres. Nesses casos, o comando strchr pode ser usado para procurar a próxima posição da cadeia com um dado caractere (por exemplo, espaço em branco ou vírgula).

```
#include <string.h>
#include <stdio.h>

#define N 50

int main()
{
    char valores[200],*vaux;
    float v[N];
    int i,n;

    /* Recebe valores como string; O primeiro numero digitado e' 'n'
```

```
      seguido dos demais numeros... */

   fgets(valores,199,stdin);
   vaux=valores;
   sscanf(vaux,"%d",&n);
   vaux=strchr(vaux,' ')+1;

   // Separa os numeros digitados no vetor 'v'
   for (i=0;i<n;i++)
   {
      sscanf(vaux,"%f",&v[i]);
      vaux=strchr(vaux,' ')+1;
   }
   // Mostra os numeros do vetor
   for(i=0;i<n;i++)
      fprintf(stdout,"%f\n",v[i]);

   return 0;
}
```

13.3 Manipulando cadeias de caracteres

Cadeias de caracteres podem ser manipuladas para diversos fins práticos. O programa abaixo ilustra algumas operações úteis.

```
#include <string.h>
#include <stdio.h>

int main()
{
   char s1[11],s2[11],s3[21];
   int resultado;

   // Le a primeira string
   fscanf(stdin,"%s",s1);
   // Le a segunda string
   fscanf(stdin,"%s",s2);

   // Verifica o comprimento das cadeias
   printf("%s tem %d caracteres\n",s1,strlen(s1));
   printf("%s tem %d caracteres\n",s2,strlen(s2));

   // Compara se sao iguais
   resultado=strcmp(s1,s2);

   if(resultado==0)
      printf("%s = %s\n",s1,s2);
   else
      if(resultado<0)
      {
         printf("%s < %s\n",s1,s2);
         // Concatena strings
         strcat(s3,s1);
         strcat(s3,s2);
         printf("%s\n",s3);
      }
      else
      {
         printf("%s > %s\n",s1,s2);
         strcat(s3,s2);
         strcat(s3,s1);
         printf("%s\n",s3);
```

```
        }
        return 0;
}
```

Observe que ‘s3’ não foi inicializado, portanto não podemos considerar que esteja limpo ou vazio. Note que no momento de apresentá-lo, algum lixo pode aparecer na tela.

Veja também as funções **strcpy** e **strncpy** para realizar cópias de caracteres e **strncat** para concatenação de n caracteres no final de uma cadeia.

Um exemplo prático é a busca de padrões em uma cadeia de caracteres. Este problema é muito comum quando pretendemos localizar palavras em um texto e padrões em uma seqüência de caracteres, como ocorre em bioinformática. O algoritmo abaixo ilustra uma solução O(nm), onde n é o comprimento da cadeia e m é o comprimento do padrão.

```
#include <stdio.h>
#include <string.h>

int main()
{
    char cadeia[101], padrao[11];
    int i,n,m;

    fscanf(stdin,"%s",padrao);
    fscanf(stdin,"%s",cadeia);

    n=strlen(cadeia);
    m=strlen(padrao);

    i=0;
    while(i<=(n-m))
    {  // compara m caracteres
       if (strncmp(&cadeia[i],padrao,m)==0)
          printf("Padrao encontrado na posicao %d\n",i);
       i++;
    }
    return 0;
}
```

13.4 Exercício:

13.4.1) Quando o padrão não é encontrado a partir da posição i, o algoritmo acima repete a busca a partir da posição i+1. Porém, quando os k primeiros caracteres são iguais aos do padrão e a diferença ocorre no caracter da posição i+k, a nova busca pode iniciar nesta posição. Note também que toda vez que o padrão for encontrado na posição i, a próxima busca pode iniciar na posição i+m. Com essas observações, escreva um algoritmo mais eficiente para buscar padrões.

14 – Ponteiros

Um ponteiro é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de outra variável na memória. Se uma variável contém o endereço de outra, então a primeira aponta para a segunda.

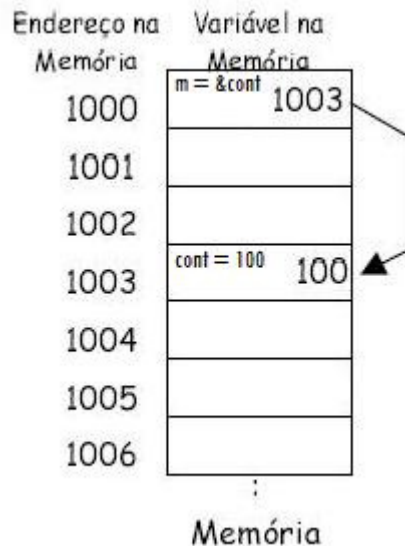



Figura 16.1: Uma variável que aponta para outra.

14.1 Declaração e manipulação de ponteiros

A forma geral de declarar uma variável do tipo ponteiro é:

```
tipo *variável;
```

O tipo base do ponteiro define que tipo de variáveis o ponteiro pode apontar. Tecnicamente o tipo de ponteiro pode apontar para qualquer lugar na memória, mas toda a aritmética de ponteiros é feita por meio do tipo base, o que requer sua declaração correta. Assim, ponteiros do tipo INT devem apontar para variáveis do tipo INT; FLOAT para FLOAT e assim por diante.

Os operadores especiais para ponteiro são * e &. O & é um operador unário que devolve o endereço de memória para onde o ponteiro indica. Neste exemplo, o trecho de código abaixo atribui a m o endereço de cont na memória.

```
int *m, cont=100;

m = &cont;
```

Já o * é o complemento de &. Podemos atribuir a outra variável o valor apontado por m. Assim teremos que q recebe o valor apontado por m. Por exemplo:

```
q = *m; // o que faz q=100
```

Existem algumas diferenças em expressões envolvendo ponteiros. São elas:

- **Atribuição:**

```
main()
{
   int x=100;
   int *p1, *p2;
   p1=&x;
   p2=p1;  // aqui p1 e p2 apontam para x
   printf("%p", p2); // escreve o endereço de x na memória e não o valor
}
```

- **Aritmética:**

Existem somente duas operações que podem ser usadas com ponteiros: soma e subtração. Considerando o trecho de código acima, temos p1 como ponteiro de inteiros, que ocupa 2 bytes. Após a expressão p1++ o ponteiro terá o endereço incrementado. Como o tipo é inteiro, o endereço será incrementado em 2 bytes. Generalizando a partir do exemplo, cada vez que um ponteiro é incrementado (ou decrementado) ele apontará para a posição de memória do próximo elemento de seu tipo base. É possível também ter a expressão "p1=p1+12" que fará p1 apontar para o próximo décimo segundo elemento do tipo base.

- **Comparação:**

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros "p" e "q", o comando a seguir é perfeitamente válido:

if (p < q) printf("p aponta para uma posição na memória mais baixa que q\n");

Geralmente, comparações de ponteiros são usadas quando dois ou mais ponteiros apontam para um objeto em comum. Como exemplo, veremos um programa que trabalha com uma pilha. Uma pilha é uma lista em que o primeiro acesso a entrar será o último a sair. É freqüentemente comparada com uma pilha de pratos em uma mesa, onde sempre se coloca ou pega um prato do topo. Para criar uma pilha são necessárias duas funções: push() e pop() (para inserir e remover elementos, respectivamente). Observe o programa abaixo:

```
/* Programa PILHA

   BASE controla o inicio da pilha e P1 a quantidade de elementos;

   Ocorre 'estouro' quando P1 se deslocar alem do limite do vetor

   (SIZE) ou quando voltar abaixo da BASE.
*/

#include <stdio.h>

#include <stdlib.h>

#define SIZE 10

int *base, *p1, pilha[SIZE];
```

```
// coloca um elemento na pilha
void push(int i)
{  p1++;
   if(p1==(base+SIZE))
   {  printf("Estouro de pilha!\n");
      exit(1);
   }
   *p1=i;
}

// retira um elemento da pilha
pop(void)
{  if(p1==base)
   {  printf("Pilha vazia!\n");
      exit(1);
   }
   p1--;
   return *(p1+1); // retorna o valor retirado
}

int main()
{  int valor;
   base=pilha; // aponta para o inicio da pilha
   p1=pilha; // inicializa p1 no inicio da pilha
   printf("\n..:: Pilha ::..");
   printf("\nDigite um numero inteiro positivo para inserir na pilha");
   printf("\nDigite zero para retirar um numero da pilha");
   printf("\nDigite -1 para sair.\n");
   do
   {  printf("Valor: ");
      scanf("%d", &valor);
      if (valor!=0)
         push(valor);
      else
         printf("O valor do topo da pilha era %d\n",pop());
   }while(valor!=-1);
}
```

Outras situações em que podemos observar o comportamento do programa quando utilizamos ponteiros:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

int *p1, *p2, x=100;
int v[SIZE]={51,24,36,49,53,62,71,87,98,105};

int main()
{  int i, pos;
   // Aritmetica de ponteiros: Observe as posicoes em memoria
   p1 = &x; // p1 aponta para x
   p2 = p1; // p2 aponta para p1
   printf("\nX = %d", x);
   printf("\nP1: Endereco[%p] Decimal[%d] Valor[%d]", p1,p1,*p1);
   printf("\nP2: Endereco[%p] Decimal[%d] Valor[%d]", p2,p2,*p2);
   p1++; // incremento de p1
   printf("\nP1: Endereco[%p] Decimal[%d] Valor[%d]", p1,p1,*p1);
   p2=p2+12;
   printf("\nP2: Endereco[%p] Decimal[%d] Valor[%d]", p2,p2,*p2);

   // Ponteiros e vetores
   p1=v; // p1 aponta para o vetor v
```

```
    printf("\n\n..:: Ponteiros e Vetores ::..");
    printf("\nDigite a posicao 1 a 10 ou -1 para sair\n");
    // Mostra o vetor inicializado
       printf("\nv = {");
       for(i=0;i<10;i++)
          printf(" %d", v[i]);
       printf(" }  <== INICIAL \n");
    do
    {  // Le uma posicao especifica
       printf("\nPosicao: ");
       scanf("%d", &pos);
       if ((pos<1)||(pos>10))
          printf("\nERRO! Digite uma posicao entre 1 e 10!\n");
       else
       {
          printf("\nO valor armazenado em v[%d] e' %d",pos,*(p1+(pos-1)));
          // Modifica o valor da posicao informada
          *(p1+(pos-1))=*(p1+(pos-1))+pos;
          // equivale a v[pos-1] = v[pos-1]+pos;
          // Mostra o vetor
          printf("\nv = {");
          for(i=0;i<10;i++)
             printf(" %d", v[i]);
          printf(" }  <== ATUALIZADO!");
       }
    }while(pos!=-1);
    return 0;
}
```

14.2 Exercício: Laboratório 7

14.2.1) Baseado nas aulas anteriores, refaça uma calculadora de vetores com 10 posições de números reais, utilizando funções e ponteiros.

14.3 Ponteiros e Vetores Multidimensionais (Matrizes)

Há uma estreita relação entre ponteiros e matrizes. Considere o seguinte fragmento de programa:

```
char str[80], *p1;
p1 = str;
```

Aqui o ‘p1’ foi inicializado com o endereço do primeiro element da matriz ‘str’. Para mostrar o quinto elemento em ‘str’, podemos fazer:

```
printf(“%c”, str[4]);
```

ou

```
printf(“%c”, *(p1+4));
```

Ambos os comandos devolvem o quinto elemento. Observe que vetores e matrizes começam em zero. Vale recordar também que o nome de um desses elementos sem índice retorna o endereço inicial da estrutura, que é o primeiro elemento.

A linguagem C fornece dois métodos para acessar elementos de matrizes: aritmética de ponteiros e indexação de matrizes, e velocidade é geralmente uma consideração em programação. Para exemplificar a diferença, duas versões da função ‘putstr()’ serão apresentadas.

```
// Indexa ‘s’ como uma matriz
void puts (char *s)
```

```
{   register int t;
    for (t=0;s[t]; t++
       putchar(s[t]);
}

// Acessa 's' como um ponteiro
void putstr(char *s)
{   while (*s)
      putchar(*s++);
}
```

Ponteirostambém podem ser organizados em matrizes como qualquer outro tipo de dado. A declaração de uma matriz de ponteiros 'int' de tamanho 10, é:

```
int *x[10];
```

Para atribuir o endereço de uma variável inteira, chamada 'var', ao terceiro elemento da matriz de ponteiros fazemos:

```
x[2] = &var;
```

Para mostrar o valor de 'var' usamos:

```
printf("%d", *x[2]);
```

Se for necessário passar uma matriz de ponteiros para uma função, pode ser usado o mesmo método que é utilizado para passar outras matrizes – simplesmente chame a função com o nome da matriz sem qualquer índice. Por exemplo, a seguinte função recebe a matriz 'x' como parâmetro:

```
void display_array(int *q[])
{  int t;
   for (t=0; t<10;t++)
      printf("%d", *q[t]);
}
```

Neste caso é importante lembrar que 'q' não é um ponteiro para inteiros; 'q' é um ponteiro para uma matriz de ponteiros para inteiros. Portanto, é necessário declarar o parâmetro 'q' como uma matriz de ponteiros para inteiros, como declarado acima.

Matrizes de ponteiros são usadas normalmente como ponteiros para strings. Assim é possível criar uma função que exiba uma mensagem de erro quando é dado seu número de código do erro, como mostrado a seguir:

```
void syntax_error(int num)
{  static char *err[] = { "Arquivo nao pode ser aberto!\n",
                          "Erro de leitura\n",
                          "Erro de escrita\n",
                          "Falha da mídia\n"
                        }
   printf("%s", err[num]);
}
```

No exemplo anterior a matriz 'err' contém ponteiros para cada string com uma mensagem de erro. O 'printf' dentro de 'syntax_error' é chamada com um ponteiro de caracteres que aponta para uma das várias mensagens de erro indexadas pelo número de erro passado para a função. Se passado o valor 1 para a função, será mostrada a mensagem 'Erro de leitura".

14.3.1 Alocação Dinâmica

É o meio pelo qual o programa pode obter memória enquanto está em execução. Variáveis globais têm o armazenamento alocado em tempo de compilação e variáveis locais usam pilha. No entanto, em tempo de execução não é possível acrescentar variáveis locais ou globais. Quando houver essa necessidade, podemos usar a memória RAM disponível para alocar conforme a necessidade.

A memória alocada pelas funções de alocação dinâmica de C é obtida do heap (região de memória livre). Embora seu tamanho seja desconhecido, o heap geralmente contém uma quantidade razoável de memória livre.

A alocação dinâmica em C baseia-se nas funções malloc(), para alocar memória e free(), para liberar a memória alocada. Apesar de existirem outras funções de alocação dinâmica, estas são as mais importantes. Elas operam em conjunto, usando a região de memória livre para estabelecer e manter a lista de armazenamento disponível. Os protótipos destas funções, que estão descritos em 'stdlib.h' são:

```
// Declaracao geral para alocar espaco
void *malloc(size_t<número de bytes>);

// Na pratica
char *p;
p = malloc(1000);
ou
// desta forma há garantia de portabilidade do código-fonte
p = malloc(50*sizeof(int));

// Declaracao geral para liberar memória
void free(void *p);

// Na pratica
free(p);
```

Após uma chamada bem sucedida de 'malloc' temos um ponteiro para o primeiro byte da região de memória alocada do heap. Se não houver memória disponível para satisfazer a requisição, a função retorna zero. Veja o trecho de código:

```
char *p1;
int *p2;
...
if (!(p1=malloc(1000)) )
{  printf("Sem memoria disponivel!\n"); exit(1); }
...
p2 = malloc(50*sizeof(int));
...
free(p1);
free(p2);
```

É importante salientar que o ponteiro enviado para a função 'free()' deve ser um ponteiro para memória alocada anteriormente por 'malloc'. Nunca se deve usar 'free()' com um argumento inválido, pois isso destruiria a lista de memória livre.

É possível operar com uma matriz alocada dinamicamente na memória através de "malloc", pois qualquer ponteiro pode ser indexado como se fosse uma matriz unidimensional. Veja no exemplo a seguir:

```
/* Aloca espaço para uma string dinamicamente, solicita a entrada do usuários e,
em seguida, mostra a string de trás para frente */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

void main(void)
{  char *s;
   register int t;
   s = malloc(80);
   if(!s)
   {  printf("Falha na solicitacao de memoria!\n"); exit(1); }
   gets(s);
   for(t=strlen(s)-1; t>0; t--)
      putchar(s[t]);
   free(s);
}
```

Acessar memória alocada como se fosse uma matriz unidimensional é simples. No entanto, matrizes dinâmicas multidimensionais levantam alguns problemas. Como as dimensões da matriz não foram definidas no programa, não é possível indexar diretamente um ponteiro como se ele fosse uma matriz multidimensional. Para conseguir uma matriz alocada dinamicamente, é necessário passar o ponteiro como um parâmetro da função. Dessa forma, a função pode definir as dimensões do parâmetro que recebe o ponteiro, permitindo, assim, a indexação normal da matriz. O exemplo a seguir constrói uma tabela dos números de 1 a 10 elevados a primeira, à segunda, à terceira e à quarta potência.

```
// Se houver alguma advertência sobre as funções table() e show(), ignore.
#include <stdio.h>
#include <stdlib.h>

int pwr(int a, int b);
void table(int p[4][10]);
void show(int p[4][10]);

void main(void)
{  int *p;
   p = (int *)malloc(40*sizeof(int));
   if (!p)
     { printf("Falha na solicitacao de memoria\n"); exit(1); }
   // aqui p e' simplesmente um ponteiro para inteiros
   table(p);
   show(p);
}

// Constroi tabela de potencias
void table(int p[4][10])
{  register int i, j;
   for(j=1;j<11;j++)
      for(i=1;i<5);i++)
         p[i-1][j-1]=pwr(j,i);
}

// Exibe tabela de potencias
void show(int p[4][10])
{  register int i,j;
   printf("%10s %10s %10s %10s\n", "N", "N^2", "N^3", "N^4");
   for(j=1;j<11;j++)
   {  for(i=1;i<5;i++)
         printf("%10d ", p[i-1][j-1]);
      printf("\n");
   }
}

// eleva um inteiro a uma potencia especificada
```

```
pwr(int a, int b)
{  register int t=1;
   for(; b; b--)
      t=t*a;
   return t;
}
```

14.4 Exercício

14.4.1) Reescreva o programa de multiplicação de matrizes e armazenamento da transposta.

15 – Funções

Em aulas anteriores vimos diversos comandos (como: printf(); scanf(); strlen(); strcat(); entre outros) que realizam tarefas mais complexas sobre valores de entrada. Esses comandos são denominados funções, pois consistem no agrupamento de instruções com uma determinada finalidade. Assim como a função main(), que agrupa as instruções do programa. Uma função é, portanto, uma seqüência de comandos que pode ser executada a partir da função principal (ou de qualquer outra função).

As funções simplificam a codificação e permitem uma melhor estruturação do programa, evitando que uma mesma seqüência de comandos seja escrita diversas vezes no corpo (escopo) da função principal. Por exemplo, suponha um programa par calcular o número C(n,p) = n! / (p!*(n-p)!) de combinações de n eventos em conjuntos de p eventos, p < n. Sem o conceito de função, teríamos que repetir três vezes as instruções para cálculo do fatorial de um número x. Com o conceito de função, precisamos apenas escrever essas instruções uma única vez e substituir x por n, p, e (n-p) para saber o resultado de cada cálculo fatorial.

```
#include <stdio.h>

double fatorial(int x); // prototipo da funcao

int main()
{  int n,p,C;
   printf("Informe n e p, com n > p: ");
   scanf("%d %d", &n, &p);
   if((p>=0)&&(n>=0)&&(p<=n))
   {  // chamada da funcao
      C = (int)(fatorial(n)/(fatorial(p)*(fatorial(n-p))));
      printf("Resultado: %d \n",C);
   }
   return 0;
}

double fatorial(int x) // escopo da funcao
{  double fat=1;
   int i;
   for(i=x; i>1; i--)
      fat=fat*i;
   return(fat);
}
```

A função fatorial() recebe o valor de uma variável da função principal, armazena em uma variável x do seu escopo, e retorna o cálculo do fatorial de x para o programa principal. As variáveis x, i e fat declaradas na função fatorial são denominadas variáveis locais desta função. Elas só existem na memória enquanto a função fatorial estiver em execução (neste exemplo, são alocadas e desalocadas três vezes na memória) e só podem ser usadas no escopo desta função. A mesma observação é válida com relação às variáveis locais n, p e C da função principal, porém essas permanecem na memória durante toda a execução do programa.

Um programa pode ter várias funções e uma função pode conter chamadas a outras funções do programa. Cada função deve ser declarada da seguinte forma:

```
tipo funcao(tipo var1, tipo var2, ..., tipo varN); /* prototipo */

tipo funcao(tipo var1, tipo var2, ..., tipo varN)
{
    /* instrucoes */
    return (valor);
```

```
}
```

O tipo do valor retornado pela função deve ser compatível com o tipo da função. O tipo void é usado quando a função não retorna valor. Os valores de entrada e saída de uma função são denominados parâmetros. Esses parâmetros podem ser de qualquer tipo válido, incluindo registros, apontadores, cadeias de caracteres, vetores, entre outros.

15.1 Parâmetros passados por valor e por referência

No caso da função fatorial, o valor de n na chamada fatorial(n) é passado para uma cópia x da variável n. Qualquer alteração em x não afeta o conteúdo de n no escopo da função principal. Dizemos então que o parâmetro é passado por valor. Isto nos permite, por exemplo, simplificar a função para:

```
double fatorial(int x)
{
   double fat=1 ;
   while(x>1)
   {   fat=fat*x ;
        x--;
   }
   return(fat);
}
```

Porém, em várias situações, desejamos alterar o conteúdo de uma ou mais variáveis no escopo da função principal. Neste caso, os parâmetros devem ser passados por referência. Isto é, a função cria uma cópia do endereço da variável correspondente na função principal em vez de uma cópia do seu conteúdo. Qualquer alteração no conteúdo deste endereço é uma alteração direta no conteúdo da variável da função principal. Por exemplo, o programa completo acima requer que p <= n. Caso contrário, podemos trocar o conteúdo dessas variáveis.

```
#include <stdio.h>

// prototipo da funcao fatorial
double fatorial(int x);

/* x e y sao apontadores para enderecos de memoria
   que guardam valores do tipo int */
void troca(int *x, int *y);

int main()
{
   int n,p,C;
   printf("Informe n e p, com n > p: ");
   scanf("%d %d", &n, &p);

   if (p>n)
      troca(&p,&n); // passa os enderecos de 'p' e 'n'

   if((p>=0)&&(n>=0))
   {
      C = (int)(fatorial(n)/(fatorial(p)*(fatorial(n-p))));
      printf("Resultado: %d \n",C);
   }

   return 0;
}

double fatorial(int x) // escopo da funcao
{
```

```
   double fat=1;
   while(x>1)
   {  fat=fat*x;
      x--;
   }
   return(fat);
}

void troca(int *x, int *y)
{
   int aux;
   // faz a troca dos valores x e y
   aux = *x;  // conteudo de x e' atribuido a aux
   *x  = *y;  // conteudo de y e' atribuido a x
   *y  = aux; // conteudo de aux e' atribuido a y
}
```

15.2 Exercício: Laboratório 8

15.2.1) Reescreva programas das aulas anteriores utilizando funções.

16 – Hierarquia de Funções

Como já foi visto em aulas anteriores, podemos chamar funções dentro da função principal (main). Após a execução de uma função, o programa sempre retorna para a posição imediatamente seguinte a sua chamada. O exemplo a seguir mostra uma calculadora de vetores utilizando ponteiros e funções.

```
#include <stdio.h>
#include <stdlib.h>
#define N 10

void mostra(float *a, char vetor)
{  int i;
   printf("\n%c = { ",vetor);
   for (i=0; i<10; i++)
      printf("%5.2f ", *(a+i));
   printf("}");
}

void calcula(float *a, float *b, float *c, int op)
{
   int i;
   switch(op)
   {   case 1: for (i=0; i<10; i++)
                *(c+i) = (*(a+i))+(*(b+i));
               break;
       case 2: for (i=0; i<10; i++)
                *(c+i) = (*(a+i))-(*(b+i));
               break;
       case 3: for (i=0; i<10; i++)
                *(c+i) = (*(a+i))*(*(b+i));
               break;
       case 4: for (i=0; i<10; i++)
               {  if((*(b+i))==0)
                    *(c+i)=0;
                else
                    *(c+i) = (*(a+i))/(*(b+i));
               }
   }
   mostra(a,'A');
   mostra(b,'B');
   mostra(c,'C');
}

void levetor(float *a, char vetor)
{
   int i;
   printf("%c = ", vetor);
   for (i=0; i<10; i++)
      scanf("%f", (a+i));
}
int main()
{
   int op,i;
   float *p1, *p2, *p3, A[10], B[10], C[10];
   p1=A;p2=B;p3=C;
   do
   {
      printf("\n..:: Calculadora de Vetores ::..");
```

```
      printf("\n\n 1 - Adicao\n 2 - Subtracao");
      printf("\n 3 - Multiplicacao\n 4 - Divisao");
      printf("\n 5 - Informar valores para vetor A");
      printf("\n 6 - Informar valores para vetor B");
      printf("\n 7 - Visualizar vetores A, B e C");
      printf("\n 8 - SAIR");
      printf("\n\nOpcao: ");
      scanf("%d", &op);
      if ((op>0)&&(op<8))
      {
       switch(op)
       {
           case 1: // Adicao
                 calcula(p1,p2,p3,op);break;
           case 2: // Subtracao
                 calcula(p1,p2,p3,op);break;
           case 3: // Multiplicacao
                 calcula(p1,p2,p3,op);break;
           case 4: // Divisao
                 calcula(p1,p2,p3,op);break;
           case 5: // le vetor A
                 levetor(p1,'A');break;
           case 6: // le vetor B
                 levetor(p2,'B');break;
           case 7: mostra(p1,'A');mostra(p2,'B');mostra(p3,'C');
       }
      }
   }while(op!=8);
   return 0;
}
```

16.1 Exercício:

A sugestão de atividade desta aula é refazer os exercícios aplicando os conceitos adquiridos até a presente aula. Pratique, por exemplo, a execução de alguns de seus programas através de um menu, onde cada chamada de programa é uma função e as funções podem ser utilizadas por mais de um programa. Exemplo: teste de número primo; sorteio de números inteiros armazenando em vetor e verificando posteriormente quais desses números armazenados são primos; conversões de temperaturas e medidas entre outras atividades.

17 – Recursão

A recursão ou recursividade na programação de computadores envolve a definição de uma função que pode invocar a si própria. Um exemplo de aplicação da recursividade pode ser encontrado nos analisadores gramaticais recursivos para linguagens de programação. A grande vantagem da recursão está na possibilidade de usar um programa de computador finito para definir, analisar ou produzir um estoque potencialmente infinito de sentenças, designs ou outros dados.

Um método comum de simplificação consiste em dividir um problema em subproblemas do mesmo tipo. Como técnica de programação, isto se denomina divisão e conquista, e constitui a chave para o desenvolvimento de muitos algoritmos importantes, bem como um elemento fundamental do paradigma de programação dinâmica. Toda função que puder ser produzida por um computador pode ser escrita como função recursiva sem o uso de iteração; reciprocamente, qualquer função recursiva pode ser descrita através de iterações sucessivas.

Um exemplo simples de recursão pode ser o seguinte: durante a leitura de um livro, uma palavra desconhecida é vista. O leitor anota o número da página (e coloca em uma pilha que até então está vazia) e pode encontrar outras palavras desconhecidas, repetindo o procedimento (acrescentando ao topo da pilha). Em algum momento do texto, o leitor encontra uma frase ou um parágrafo onde está a última palavra anotada e pelo contexto da frase, descobre o seu significado. Então o leitor volta para a página anterior e continua lendo dali. Paulatinamente, remove-se seqüencialmente cada anotação que está no topo da pilha. Finalmente, o leitor volta para a sua leitura já sabendo o significado da(s) palavra(s) desconhecida(s). Isto é uma forma de recursão.

Para aplicar essa idéia em nossos programas, considere como problema o cálculo da soma de números inteiros num intervalo [m,n], onde ‘m’ e ‘n’ pertencem a Z e ‘m’ é menor igual a ‘n’. A solução iterativa é: m + (m+1) + (m+2) + ... + n, que pode ser implementada com mostra a função a seguir:

```
int Soma(int m, int n)
{
   int soma=0, i;

   for (i=m; i<=n; i++)
      soma += i;

   return(soma);
}
```

A solução recursiva está diretamente ligada ao conceito de indução matemática. A **indução fraca** usa como hipótese que a solução de um problema de tamanho t pode ser obtida a partida de sua solução de tamanho t-1. Por exemplo, se soubermos o resultado da soma de ‘m’ até ‘n-1’, basta somar n a este resultado. Soma(m,n) = Soma(m,n-1) + n. Neste caso, dizemos que a recursão é **decrescente**. Podemos também obter o mesmo resultado somando ‘m’ com a soma de ‘m+1’ até n: Soma(m,n) = m + Soma(m+1,n). Neste caso, dizemos que a recursão é **crescente**. A solução recursiva, portanto, pode ser implementada com funções do tipo:

```
tipo funcao (<parametros>)
{
   <variaveis locais>

   if (<condicao de parada>)
   {
      <comandos finais>
      return(valor);
   } else {
```

```
      <comando iniciais>
      <chamada recursiva>
      <comandos finais>
      return(valor);
   }
}
```

Com base na função utilizada como exemplo temos:

- Solução Crescente

```
int Soma(int m, int n)
{  if (m==n) {
      return(n);
   } else {
      return(m+Soma(m+1,n));
   }
}
```

- Solução Decrescente

```
int Soma(int m, int n)
{  if (m==n) {
      return(m);
   } else {
      return(Soma(m,n-1)+n);
   }
}
```

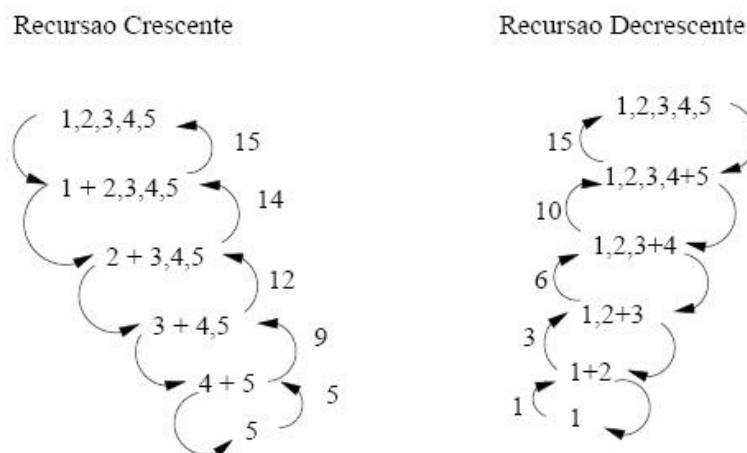



Figura 1: Árvores de Recursão para m=1 e n=5.

Observe que nestes exemplos não precisamos de variáveis locais, nem de comando iniciais e finais, portanto, podemos colocar a chamada recursiva no comando de retorno. As árvores de recursão mostradas na figura 1 ajudam a visualizar o comportamento dessas funções.

A **indução forte** usa outra hipótese: a solução de um problema de tamanho t depende de suas soluções de tamanhos t', para todo t' < t. Esta estratégia é denominada **divisão e conquista**. Por exemplo, podemos dizer que Soma(m,n) = Soma(m, ((m+n)/2)) + Soma((((m+n)/2) +1), n). Ou seja, o problema é dividido em subproblemas similares (divisão), onde são resolvidos recursivamente (chegando ao tamanho mínimo, o problema é resolvido de forma direta), e as soluções dos subproblemas são combinadas para gerar a solução do problema de tamanho maior (conquista). Isto requer, porém, ao menos duas chamadas recursivas:

```
tipo funcao(<parametros>)
```

```
{
   <variaveis locais>

   if (<condicao de parada>)
   {
      <comandos finais>
      return(valor);
   } else {
      <comandos iniciais>
      <primeira chamada recursiva>
      <segunda chamada recursiva>
      <comandos finais>
      return(valor);
   }
}
```

Com base no exemplo acima, temos:

```
int Soma(int m, int n)
{
   if (m==n) {
      return(m);
   } else {
      return(Soma(m,(m+n)/2+Soma((m+n)/2+1,n));
   }
}
```

Mais uma vez, pelas mesmas razões acima, as chamadas recursivas podem ser feitas no comando de retorno. A árvore de recursão é apresentada na figura 2. Observe que em todos os exemplos, as chamadas recursivas são para a própria função. Dizemos então que a **recursão é direta**. Em alguns problemas, porem, uma função A pode chamar uma função B que chama C, que chama outra função e que ao final chama A. Neste caso dizemos que a **recursão é indireta**.

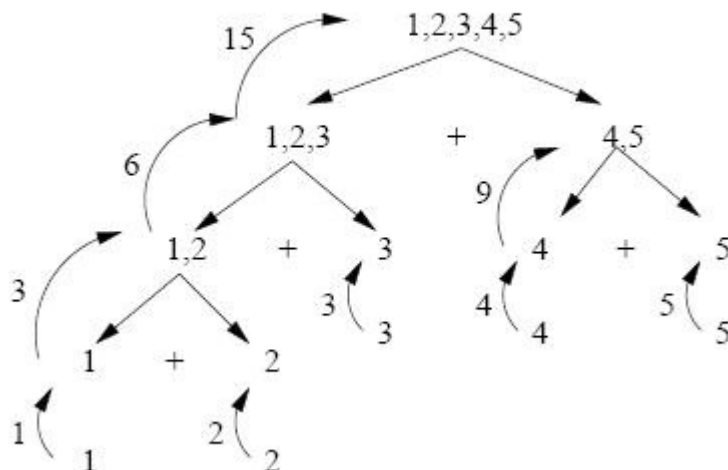



Figura 2: Árvore de recursão para m=1 e n=5.

17.1 Exercícios: Laboratório 9

17.1.1) Escreva uma função recursiva para calcular o fatorial de um número inteiro n. Note que o fatorial de n pode ser definido de forma recursiva como:

$$fat(n) \begin{cases} 1 & \text{se } n = 0 \\ n \times fat(n-1) & \text{se } n > 0 \end{cases}$$

17.1.2) Os termos de uma sequencia de Fibonacci (exemplo: 0,1,1,2,3,5,8,...) são definidos de forma recursiva. Escreva uma função recursiva para retornar o n-ésimo termo desta sequencia.

$$fib(n) = \begin{cases} n - 1 & \text{se } 1 \leq n \leq 2 \\ fib(n-1) + fib(n-2) & \text{se } n > 2 \end{cases}$$

17.2 Recursão (Ordenação)

Nesta aula continuaremos a tratar sobre o tema de recursão (ou recursividade) e recorrendo a abordagem prévia sobre ordenação.

17.2.1 Ordenação por indução fraca

Os algoritmos de ordenação de sequências por seleção, inserção e permutação podem ser implementados usando indução fraca. Na ordenação por inserção, ordenamos a sequência até a penúltima posição e inserimos o último elemento na posição correta da sequência ordenada. Na ordenação por seleção, selecionamos o maior elemento, colocamos ele na última posição e depois repetimos o processo para s subseqüência terminada no penúltimo. Na ordenação por permutação, o processo é similar. Os elementos são permutados até que o maior seja o último e depois repetimos o processo para a subseqüência terminada no penúltimo.

Por exemplo, a função abaixo ordena de forma recursiva um vetor 'v' de inteiros e de tamanho 'n' por inserção.

```
void Insercao(int *v, int n)
{
   int i,j;
   /* Condicao de parada, n == 1;
      Retorna a chamada recursiva com (v,n==2), atribui i, j e faz trocas.
   */
   if (n > 1)
   {
      Insercao(v,n-1);
      i=n-2;
      j=n-1;
      while((i>=0) && (v[i] > v[j]))
      {
         troca(&v[i],&v[j]);
         i--;
         j--;
      }
   }
}
```

17.2.2 Ordenação por indução forte

A vantagem da indução forte é reduzir a complexidade da ordenação de O(n2) para O(n log n). O algoritmo mais simples nesta linha é o merge – sort. Este algoritmo subdivide a sequência em duas, ordena de forma recursiva cada parte e depois intercala as partes ordenadas.

```
/* Considerando que o vetor esta ordenado do inicio ate o meio e do meio +1
   ate' o final, intercala seus elementos para que fique ordenado do inicio
   ao fim
*/

void Intercala(int *v, int inicio, int meio, int fim)
{
   // Ordenacao requer memoria auxiliar do mesmo tamanho da entrada
   int i,j,k,aux[N];

   i=inicio;
   j=meio+1;
   k=inicio;

   while((i<=meio)&&(j<=fim))
   {
      if(v[i]<=v[j])
      {
         vaux[k]=v[i];
         i++;k++;
      } else {
         vaux[k]=v[j];
         j++;k==;
      }
   }

   for(i=i; i <= meio; i++,k++)
      vaux[k]=v[i];

   for(j=j; j <= meio; j++,k++)
      vaux[k]=v[j];

   // Copia de volta para 'v'
   for(i=inicio; i <= fim; i++)
      v[i]=vaux[i];
}

void MergeSort(int *v, int inicio, int fim)
{
   int meio;

   if(inicio<fim)
   {
     meio=(inicio+fim)/2;
     MergeSort(v,inicio,meio);
     MergeSort(v,meio+1,fim);
     Intercala(v,inicio,meio,fim);
   }
}
```

Uma desvantagem do algoritmo acima, porém, é a necessidade de memória auxiliar na função de intercalação, do mesmo tamanho da entrada. Isto pode ser um problema para sequências muito grandes.

Outro algoritmo que usa indução forte, tem complexidade O(n log n) no caso médio, e O(n2) no pior caso, mas não requer memória auxiliar é o quick-sort. Este algoritmo particiona a sequência

em duas parte de tal forma que todos os elementos da primeira parte são menores ou iguais aos da segunda. A sequência é ordenada repetindo-se este processo de forma recursiva para cada parte.

```
void QuickSort(int *v, int inicio, int fim)
{
   int p;

   if(inicio<fim)
   {
      p=Particiona(v,inicio,fim);
      QuickSort(v,inicio,p);
      QuickSort(v,p+1,fim);
   }
}
```

17.3 Exercícios

17.3.1) Escreva as funções de ordenação de forma recursiva, por seleção e por permutação, de um vetor ‘v’ com ‘n’ elementos.

17.3.2) Escreva uma função recursiva para ordenar ‘v’ por partição (ou seja, complete o código fonte do quick-sort).

18 – Registros

Variáveis compostas são aquelas que agrupam um certo número de elementos em um único identificador. No caso de vetores e matrizes, todos os elementos agrupados são do mesmo tipo e, portanto, dizemos que são variáveis compostas homogêneas.

Em muitas situações, porém, desejamos agrupar variáveis de tipos diferentes. Um exemplo é o caso em que agrupamos dados sobre uma pessoa e/ou objeto (ex: RA, nome, curso, notas de um aluno). O conceito de registro permite criar um identificador único associado a todos esses dados. Portanto um registro é uma variável composta heterogênea e seus elementos são chamados campos.

Diferentes tipos de registro podem ser criados com campos diferentes. Parar definir um tipo de registro, usamos o comando **`typedef struct`**. O programa abaixo define um tipo de registro, Aluno, uma variável desse tipo, armazena dados nos seus campos e depois imprime os dados armazenados.

```
#include <stdio.h>
#include <string.h>

typedef struct _aluno
{
   int   RA;
   char  nome[50];
   float nota[3];
} Aluno;

int main()
{
 Aluno a; // variavel 'a' do tipo Aluno

 a.RA = 909090;
 strcpy(a.nome,"Carlos Silvestre");
 a.nota[0]= 3.0;
 a.nota[1]=10.0;
 a.nota[2]= 7.0;
 printf("%6d  %s  %5.2f  %5.2f  %5.2f\n",  a.RA,  a.nome,  a.nota[0],  a.nota[1],
a.nota[2]);
 return 0;
}
```

Observe que cada campo do registro é uma variável de qualquer tipo válido, incluindo um outro registro, vetor, matriz, etc.

```
#include <stdio.h>
typedef struct _ponto {
   float x;
   float y;
} Ponto;

typedef struct _reta {
   Ponto p1;
   Ponto p2;
} Reta;

// Pontos consecutivos sao interligados por segmento de reta
typedef struct _curva {
   Ponto pt[100];
   int npts;
```

```
} Curva;

int main()
{
   Reta r;
   Curva c;
   int i;

   // Ler os pontos da reta
   scanf("%f %f", &r.p1.x, &r.p1.y);
   scanf("%f %f", &r.p2.x, &r.p2.y);

   // Ler os pontos da curva
   scanf("%d", &c.npts);
   for (i=0; i < c.npts; i++)
      scanf("%f %f", &c.pt[i].c, &c.pt[i].y);

   /* Complete o programa para que ele verifique se
      existe interseccao entre a curva e a reta */

   return 0;
}
```

Vetores de registros podem ser usados para armazenar base de dados (ou parte da base) em memória. O programa abaixo ilustra o armazenamento de uma mini base com 5 nomes e 5 telefones.

```
#include <stdio.h>
#include <string.h>

typedef struct _agenda {
   char nome[50];
   int  telefone;
} Agenda;

int main()
{  Agenda amigo[5];
   char nome_aux[50];
   int i, comp;

   printf("Entre com os nomes:\n");
   for (i=0; i<5; i++)
   {
      fgets(nome_aux,49,stdin);
      comp=strlen(nome_aux);
      strncpy(amigo[i].nome, nome_aux, comp-1); // elimina \n
      amigo[i].nome[comp-1]= '\0'; // insere \0 por garantia
   }
   printf("Entre com os telefones:\n");
   for (i=0; i<5; i++)
      fscanf(stdin,"%d", &amigo[i].telefone);

   for (i=0; i<5; i++)
      fprintf(stdout,"%s: %d\n", amigo[i].nome, amigo[i].telefone);

   return 0;
}
```

18.1 Exercício: Laboratório 10

18.1.1) Faça um programa para armazenar 20 nomes e 20 telefones em um vetor de registros, colocar os elementos do vetor em ordem crescente de nome, e depois imprimir o vetor ordenado.

19 – Listas

Para representarmos um grupo de dados, já vimos que podemos usar um vetor em linguagem C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000
int vet[MAX];
```

Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura 1:

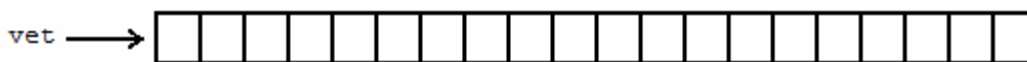



Figura 1: Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado, indexando-se o ponteiro para o primeiro elemento.

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo `vet`, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de `vet` é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação `vet[i]`. Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais **estruturas** são chamadas **dinâmicas** e armazenam cada um dos seus elementos usando alocação dinâmica.

Uma lista é uma sequência dinâmica de elementos, denominados nós. Cada nó é uma estrutura que possui campos e ponteiros, conforme a aplicação desenvolvida. Podemos, por exemplo, usar uma lista para armazenar elementos de um conjunto, que podem ser inseridos e removidos do conjunto durante a execução do programa. Neste sentido, listas são dinâmicas (possuem tamanho variável durante a execução do programa) pois seus nós são alocados dinamicamente em memória e interligados por apontadores. Se cada nó da lista possuir um apontador para o nó seguinte, a lista é dita **ligada simples** (ou encadeada). Cada nó pode ainda manter um apontador para o nó anterior, além do apontador para o próximo nó. Neste caso, a lista é dita **ligada dupla** (ou duplamente encadeada).

19.1 Lista encadeada

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os

elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A figura 2 ilustra o arranjo da memória de uma lista encadeada.

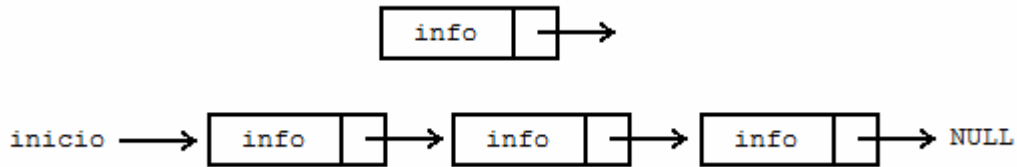



Figura 2: Exemplo de um nó e do arranjo de memória de uma lista encadeada.

A estrutura consiste numa seqüência encadeada de elementos, em geral chamados de nós da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó). Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, sinalizando que não existe um próximo elemento.

Para exemplificar a implementação de listas encadeadas em C, vamos considerar um exemplo simples em que queremos armazenar valores inteiros numa lista encadeada.

```
struct lista {
   int info;
   struct lista* prox;
};

typedef struct lista Lista;
```

Devemos notar que trata-se de uma estrutura auto-referenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Embora não seja essencial, é uma boa estratégia definirmos o tipo Lista como sinônimo de struct lista, conforme ilustrado acima. O tipo Lista representa um nó da lista e a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo Lista*).

Considerando a definição de Lista, podemos definir as principais funções necessárias para implementar uma lista encadeada.

19.1.1 Função de inicialização

A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro NULL, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é NULL. Uma possível implementação da função de inicialização é mostrada a seguir:

```
/* função de inicialização: retorna uma lista vazia */

Lista* inicializa (void)
{
   return NULL;
}
```

19.1.2 Função de inserção

Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
/* inserção no início: retorna a lista atualizada */
Lista* insere (Lista* l, int i)
{
   Lista* novo = (Lista*) malloc(sizeof(Lista));
   novo->info = i;
   novo->prox = l;
   return novo;
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento. A figura 3 ilustra a operação de inserção de um novo elemento no início da lista.

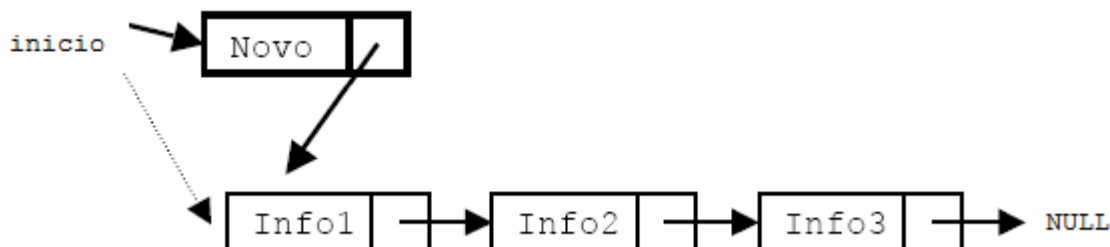



Figura 3: Inserção de novo elemento na lista.

A seguir, ilustramos um trecho de código que cria uma lista inicialmente vazia e insere nela novos elementos.

```
int main (void)
{
   Lista* l; /* declara uma lista não inicializada */
   l = inicializa(); /* inicializa lista como vazia */
   l = insere(l, 23); /* insere na lista o elemento 23 */
   l = insere(l, 45); /* insere na lista o elemento 45 */
   ...
   return 0;
}
```

Observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção de um novo elemento.

19.1.3 Função que percorre os elementos da lista

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.

```
/* função imprime: imprime valores dos elementos */
void imprime (Lista* l)
{
   Lista* p; /* variável auxiliar para percorrer a lista */
   for (p = l; p != NULL; p = p->prox)
   printf("info = %d\n", p->info);
}
```

19.1.4 Função que verifica se lista está vazia

Pode ser útil implementarmos uma função que verifique se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é NULL. Uma implementação dessa função é mostrada a seguir:

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
   if (l == NULL)
      return 1;
   else
      return 0;
}
```

Essa função pode ser re-escrita de forma mais compacta, conforme mostrado abaixo:

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
   return (l == NULL);
}
```

19.1.5 Função de busca

Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

```
/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{
   Lista* p;
   for (p=l; p!=NULL; p=p->prox)
      if (p->info == v)
         return p;
   return NULL; /* não achou o elemento */
}
```

19.1.6 Função que retira um elemento da lista

Para completar o conjunto de funções que manipulam uma lista, devemos implementar uma função que nos permita retirar um elemento. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. As figuras 4 e 5 ilustram as operações de remoção.

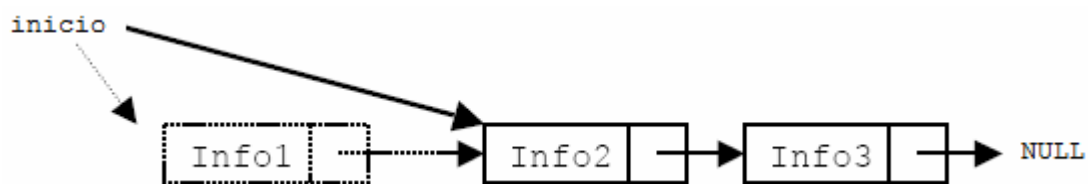



Figura 4: Remoção do primeiro elemento da lista.

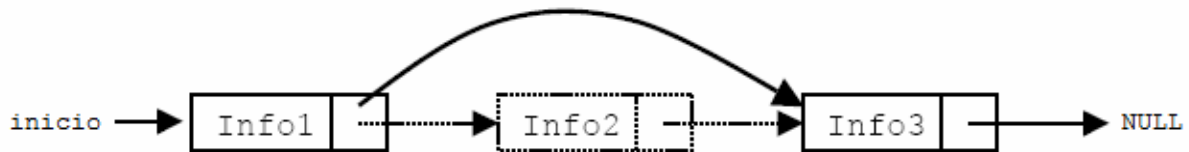



Figura 5: Remoção de um elemento do meio da lista.

Uma possível implementação da função para retirar um elemento da lista é mostrada a seguir. Inicialmente, busca-se o elemento que se deseja retirar, guardando uma referência para o elemento anterior.

```
/* função retira: retira elemento da lista */
Lista* retira (Lista* l, int v)
{
   Lista* ant = NULL; // ponteiro para elemento anterior
   Lista* p = l; // ponteiro para percorrer a lista
   /* procura elemento na lista, guardando anterior */
   while (p != NULL && p->info != v)
   {
      ant = p;
      p = p->prox;
   }
   // verifica se achou elemento
   if (p == NULL)
      return l; // não achou: retorna lista original
   // retira elemento
   if (ant == NULL)
   {
```

```
      l = p->prox; // retira elemento do inicio
   }
   else
   {
      ant->prox = p->prox; // retira elemento do meio da lista
   }
   free(p);
   return l;
}
```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, conforme pode ser observado na implementação acima. Mais adiante, estudaremos a implementação de filas com listas encadeadas. Numa fila, devemos armazenar, além do ponteiro para o primeiro elemento, um ponteiro para o último elemento. Nesse caso, se for removido o último elemento, veremos que será necessário atualizar a fila.

19.1.7 Função para liberar a lista

Uma outra função útil que devemos considerar destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada abaixo. A função percorre elemento a elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

```
void libera (Lista* l)
{
   Lista* p = l;
   while (p != NULL)
   {
      Lista* t = p->prox; // guarda referência para o próximo elemento
      free(p); // libera a memória apontada por p
      p = t; // faz p apontar para o próximo
   }
}
```

Um programa que ilustra a utilização dessas funções é mostrado a seguir.

```
#include <stdio.h>
int main (void)
{
   Lista* l;           // declara uma lista não iniciada
   l = inicializa();   // inicia lista vazia
   l = insere(l, 23); // insere na lista o elemento 23
   l = insere(l, 45); // insere na lista o elemento 45
   l = insere(l, 56); // insere na lista o elemento 56
   l = insere(l, 78); // insere na lista o elemento 78
   imprime(l);         // mostra elementos; imprimirá: 78 56 45 23
   l = retira(l, 78); // retira o elemento 78 da lista
   imprime(l);         // mostra elementos; imprimirá: 56 45 23
   l = retira(l, 45); // retira o elemento 45 da lista
   imprime(l);         // mostra elementos; imprimirá: 56 23
   libera(l);          // libera memória utilizada
   return 0;
}
```

Mais uma vez, observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção e a cada remoção de um elemento. Esquecer de atribuir o valor de retorno à variável que representa a lista pode gerar erros graves. Se, por exemplo, a função retirar o primeiro elemento da lista, a variável que representa a lista, se não fosse atualizada, estaria apontando para um nó já liberado. Como alternativa, poderíamos fazer com que as funções insere e retira recebessem o endereço da variável que representa a lista. Nesse caso, os parâmetros das funções seriam do tipo ponteiro para lista (Lista** l) e seu conteúdo poderia ser acessado/atualizado de dentro da função usando o operador conteúdo (*l).

19.2 Exercício:

19.2.1) Faça um programa que crie uma lista e nela permita: inserir, excluir, buscar, ordenar.

19.3 Listas Genéricas

Um nó de uma lista encadeada possui basicamente duas informações: o encadeamento e a informação armazenada. Logo, podemos ter uma informação simples como um número inteiro, um número real ou outra estrutura maior ou mais complexa. Dessa forma é possível construir listas heterogêneas, ou seja, cada nó da lista pode conter dados distintos.

Um exemplo da aplicação de lista heterogênea seria para calcular áreas de objetos geométricos planos:

r = b * h   t = (b * h) / 2   c = pi * (r * r)

Devemos definir um tipo para cada objeto a ser representado:

```
struct retangulo {
   float b;
   float h;
};
typedef struct retangulo Retangulo;

struct triangulo {
   float b;
   float h;
};
typedef struct triangulo Triangulo;

struct circulo {
   float r;
};
typedef struct circulo Circulo;
```

O nó da lista deve ser composto por três campos:

- um identificador de qual objeto está armazenado no nó
- um ponteiro para a estrutura que contém a informação
- um ponteiro para o próximo nó da lista

É importante salientar que, a rigor, a lista é homogênea, no sentido de que todos os nós contêm as mesmas informações. O ponteiro para a informação deve ser do tipo genérico, pois não sabemos a princípio para que estrutura ele irá apontar: pode apontar para um retângulo, um triângulo ou um círculo. Um ponteiro genérico em C é representado pelo tipo void*. A função do tipo "ponteiro genérico" pode representar qualquer endereço de memória, independente da

informação de fato armazenada nesse espaço. No entanto, de posse de um ponteiro genérico, não podemos acessar a memória por ele apontada, já que não sabemos a informação armazenada. Por esta razão, o nó de uma lista genérica deve guardar explicitamente um identificador do tipo de objeto de fato armazenado. Consultando esse identificador, podemos converter o ponteiro genérico no ponteiro específico para o objeto em questão e, então, acessarmos os campos do objeto.

Como identificador de tipo, podemos usar valores inteiros definidos como constantes simbólicas:

```
#define RET 0
#define TRI 1
#define CIR 2
```

Assim, na criação do nó, armazenamos o identificador de tipo correspondente ao objeto sendo representado. A estrutura que representa o nó pode ser dada por:

```
/* Define o nó da estrutura */
struct listagen {
   int tipo;
   void *info;
   struct listagen *prox;
};
typedef struct listagen ListaGen;
```

A função para a criação de um nó da lista pode ser definida por três variações, uma para cada tipo de objeto que pode ser armazenado.

```
// Cria um nó com um retângulo, inicializando os campos base e altura
ListaGen* cria_ret (float b, float h)
{  Retangulo* r;
   ListaGen* p;
   // aloca retângulo
   r = (Retangulo*) malloc(sizeof(Retangulo));
   r->b = b;
   r->h = h;
   // aloca nó
   p = (ListaGen*) malloc(sizeof(ListaGen));
   p->tipo = RET;
   p->info = r;
   p->prox = NULL;
   return p;
}


// Cria um nó com um triângulo, inicializando os campos base e altura
ListaGen* cria_tri (float b, float h)
{  Triangulo* t;
   ListaGen* p;
   // aloca triângulo
   t = (Triangulo*) malloc(sizeof(Triangulo));
   t->b = b;
   t->h = h;
   // aloca nó
   p = (ListaGen*) malloc(sizeof(ListaGen));
   p->tipo = TRI;
   p->info = t;
   p->prox = NULL;
   return p;
}
```

```
// Cria um nó com um círculo, inicializando o campo raio
ListaGen* cria_cir (float r)
{  Circulo* c;
   ListaGen* p;
   // aloca círculo
   c = (Circulo*) malloc(sizeof(Circulo));
   c->r = r;
   // aloca nó
   p = (ListaGen*) malloc(sizeof(ListaGen));
   p->tipo = CIR;
   p->info = c;
   p->prox = NULL;
   return p;
}
```

Uma vez criado o nó, podemos inseri-lo na lista como já vínhamos fazendo com nós de listas homogêneas.

Para calcular devidamente as áreas, utilizaremos uma função auxiliar.

```
#define PI 3.14159

// Funcao Auxiliar: calcula area correspondente ao nó
float area (ListaGen *p)
{  float a; /* área do elemento */
   switch (p->tipo)
   {      case RET: { // converte para retângulo e calcula área
                      Retangulo *r = (Retangulo*) p->info;
                      a = r->b * r->h; } break;
          case TRI: { // converte para triângulo e calcula área
                      Triangulo *t = (Triangulo*) p->info;
                      a = (t->b * t->h) / 2; } break;
          case CIR: { // converte para círculo e calcula área
                      Circulo *c = (Circulo)p->info;
                      a = PI * c->r * c->r; } break;
   }
   return a;
}

// Funcao para calculo da maior area
float maior_area (ListaGen* l)
{
   float maior = 0.0; // maior area
   ListaGen* p;
   for (p=l; p!=NULL; p=p->prox)
   {
      float a = area(p); // area do no
      if (a > maior)
         maior = a;
   }
   return maior;
}
```

Outra possibilidade é a implementação de funções específicas para o cálculo de cada objeto, a critério do desenvolvedor. Quando utilizamos ponteiros genéricos e específicos devemos estar mais atentos para evitar equívocos, uma vez que o compilador não é capaz de verificar se as conversões realizadas são válidas.

19.4 Listas Circulares

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, as arestas que delimitam uma face podem ser agrupadas por uma estrutura circular. Para esses casos, podemos usar listas circulares.

Numa lista circular, o último elemento tem como próximo o primeiro elemento da lista, formando um ciclo. A rigor, neste caso, não faz sentido falarmos em primeiro ou último elemento. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista. A figura 1 ilustra o arranjo da memória para a representação de uma lista circular.

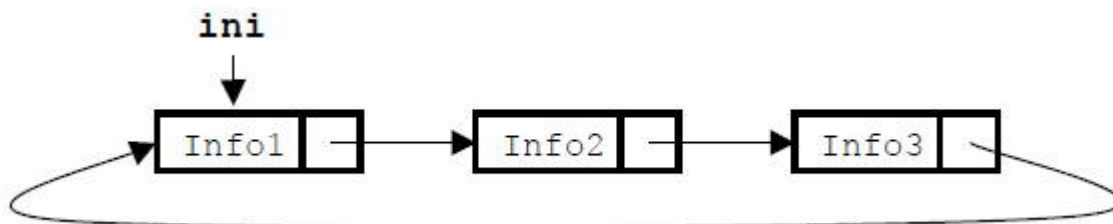



Figura 1: arranjo da memória em uma lista circular

Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançarmos novamente esse mesmo elemento. O código abaixo exemplifica essa forma de percorrer os elementos. Neste caso, para simplificar, consideramos uma lista que armazena valores inteiros. Devemos salientar que o caso em que a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um elemento inicial vale NULL).

```
void imprime_circular (Lista* l)
{
   Lista* p = l; // faz p apontar para o nó inicial
   // Verifica se a lista nao esta vazia
   if (p)
   {  // percorre os elementos ate retornar ao inicio
      do
      {  printf("%d\n", p->info); // imprime informação do nó
         p = p->prox; // avança para o próximo nó
      } while (p != l);
}
```

A lista circular também pode ser duplamente encadeada, como veremos a seguir.

19.5 Lista Duplamente Encadeada

A estrutura de lista encadeada vista anteriormente caracteriza-se por formar um encadeamento simples entre os elementos: cada elemento armazena um ponteiro para o próximo elemento da lista. Desta forma, não temos como percorrer eficientemente os elementos em ordem inversa, isto é, do final para o início da lista.

O encadeamento simples também dificulta a retirada de um elemento da lista. Mesmo se tivermos o ponteiro do elemento que desejamos retirar, temos que percorrer a lista, elemento por elemento, para encontrarmos o elemento anterior, pois, dado um determinado elemento, não temos como acessar diretamente seu elemento anterior.

Para solucionar esses problemas, podemos formar o que chamamos de listas duplamente encadeadas. Nelas, cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Desta forma, dado um elemento, podemos acessar ambos os elementos adjacentes: o anterior e o próximo. Se tivermos um ponteiro para o último elemento da lista,

podemos percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior, até alcançar o primeiro elemento da lista (cujo anterior é NULL).

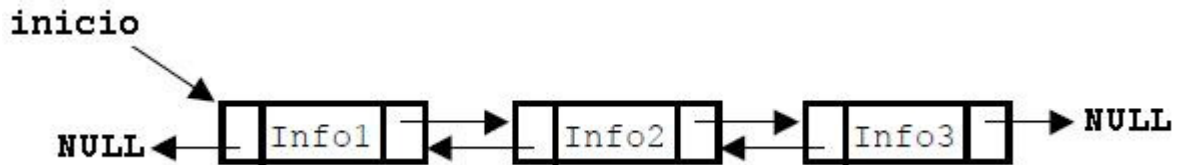



Figura 2: arranjo de memória para uma lista duplamente encadeada

Para exemplificar a implementação de listas duplamente encadeadas, vamos novamente considerar o exemplo simples no qual queremos armazenar valores inteiros na lista. O nó da lista pode ser representado pela estrutura abaixo e a lista pode ser representada através do ponteiro para o primeiro nó.

```
struct lista2 {
   int info;
   struct lista2* ant;
   struct lista2* prox;
};
typedef struct Lista2 Lista2;
```

Com base nas definições acima, exemplificamos a seguir a implementação de algumas funções que manipulam listas duplamente encadeadas.

19.5.1 Função de inserção

O código a seguir mostra uma possível implementação da função que insere novos elementos no início da lista. Após a alocação do novo elemento, a função acertar o duplo encadeamento.

```
/* inserção no início */
Lista2* insere (Lista2* l, int v)
{
   Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
   novo->info = v;
   novo->prox = l;
   novo->ant = NULL;
   // verifica se lista não está vazia
   if (l != NULL)
      l->ant = novo;
   return novo;
}
```

Nessa função, o novo elemento é encadeado no início da lista. Assim, ele tem como próximo elemento o antigo primeiro elemento da lista e como anterior o valor NULL. A seguir, a função testa se a lista não era vazia, pois, neste caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento. De qualquer forma, o novo elemento passa a ser o primeiro da lista, e deve ser retornado como valor da lista atualizada.

19.5.2 Função de busca

A função de busca recebe a informação referente ao elemento que queremos buscar e tem como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

```
/* função busca: busca um elemento na lista */
Lista2* busca (Lista2* l, int v)
{
   Lista2* p;
   for (p=l; p!=NULL; p=p->prox)
   if (p->info == v)
   return p;
   return NULL; /* não achou o elemento */
}
```

19.5.3 Função que retira um elemento da lista

A função de remoção fica mais complicada, pois temos que acertar o encadeamento duplo. Em contrapartida, podemos retirar um elemento da lista conhecendo apenas o ponteiro para esse elemento. Desta forma, podemos usar a função de busca acima para localizar o elemento e em seguida acertar o encadeamento, liberando o elemento ao final.

Se p representa o ponteiro do elemento que desejamos retirar, para acertar o encadeamento devemos conceitualmente fazer:

p->ant->prox = p->prox;
p->prox->ant = p->ant;

isto é, o anterior passa a apontar para o próximo e o próximo passa a apontar para o anterior.

Quando p apontar para um elemento no meio da lista, as duas atribuições acima são suficientes para efetivamente acertar o encadeamento da lista. No entanto, se p for um elemento no extremo da lista, devemos considerar as condições de contorno. Se p for o primeiro, não podemos escrever p->ant->prox, pois p->ant é NULL; além disso, temos que atualizar o valor da lista, pois o primeiro elemento será removido.

Uma implementação da função para retirar um elemento é mostrada a seguir:

```
/* função retira: retira elemento da lista */
Lista2* retira (Lista2* l, int v)
{
   Lista2* p = busca(l,v);
   if (p == NULL)
      return l; /* não achou o elemento: retorna lista inalterada */
   /* retira elemento do encadeamento */
   if (l == p)
      l = p->prox;
   else
      p->ant->prox = p->prox;
   if (p->prox != NULL)
      p->prox->ant = p->ant;
   free(p);
   return l;
}
```

19.6 Exercício: Laboratório 11

19.6.1) Faça um programa que crie uma lista duplamente encadeada e nela permita: inserir, excluir, buscar, ordenar.

20 – Arquivos

Até o momento, a entrada padrão (teclado) e a saída padrão (monitor) foram utilizadas para a comunicação de dados. Muitas situações, porém, envolvem uma grande quantidade de dados, difícil de ser digitada e exibida na tela. Esses dados são armazenados em arquivos u ficam em dispositivos secundários de memória (como disco rígido ou pendrive, por exemplo). Portanto, um arquivo é uma seqüência de bytes com uma identificação que permite o sistema operacional exibir na tela seu nome, tamanho e data de criação.

Existem dois tipos de arquivos: texto e binário. Um arquivo texto armazena os dados em ASCII, na forma de seqüência de caracteres, os quais podem ser exibidos na tela com o comando **`more`** (em Linux/Unix) ou **`type`** (no DOS ou em prompt de comando do Windows). Já um arquivo binário armazena os dados na forma de binária (seqüência de bits), e normalmente ocupa bem menos espaço em memória do que um arquivo texto para armazenar a mesma informação.

Os arquivos são manipulados com variáveis do tipo apontador para arquivo, as quais são declaradas da seguinte forma:

```
#include <stdio.h>

int main()
{
   FILE *ponteiro_de_arquivo;

   return 0;
}
```

Um ponteiro de arquivo é um ponteiro para uma área na memória (buffer) onde estão contidos vários dados sobre o arquivo a ler ou escrever, tais como o nome do arquivo, estado e posição corrente. O buffer apontado pelo ponteiro de arquivo é a área intermediária entre o arquivo e o programa.

Este buffer intermediário entre o arquivo e o programa é chamado ‘fluxo’, e no jargão dos programadores é comum falar em funções que operam fluxos. Isto se deve ao fato de que fluxo é uma entidade lógica genérica, que pode estar associada a uma unidade (disco, porta serial, entre outras).

20.1 Funções mais comuns do sistema de arquivo

```
---------------------------------------------------------------------------
Função      Operação
---------------------------------------------------------------------------

fopen()     Abre um fluxo
fclose()    Fecha um fluxo
putc()      Escreve um caractere para um fluxo
getc()      Lê um caractere do fluxo
fseek()     Procura por um byte específico no fluxo
fprintf()   É para um fluxo o equivalente ao printf() no console
fscanf()    É para um fluxo o equivalente ao scanf() no console
feof()      Retorna verdadeiro se o fim do arquivo é encontrado
ferror()    Retorna verdadeiro se ocorreu um erro
fread()     Lê um bloco de dados do fluxo
fwrite()    Escreve um bloco de dados para um fluxo
rewind()    Reposiciona o localizador de posição para o início do arquivo
remove()    Apaga um arquivo
---------------------------------------------------------------------------
```

20.2 Abrindo um Arquivo

A função **fopen()** serve a dois propósitos. Primeiro, ela abre um fluxo para uso e liga um arquivo com ele. Segundo, retorna o ponteiro de associação àquele arquivo. Mais freqüentemente, e para o resto desta discussão, considere que arquivo é um arquivo em disco.

A função fopen() tem este protótipo:

```
FILE *fopen(char *nome_de_arquivo, char *modo);
```

onde modo é uma string contendo o estado desejado para abertura. O nome do arquivo deve ser uma string de caracteres que compreende um nome de arquivo válido para o sistema operacional e onde possa ser incluída uma especificação de caminho (PATH).

Como determinado, a função fopen() retorna um ponteiro de arquivo que não deve ter o valor alterado pelo seu programa. Se um erro ocorre quando se está abrindo um arquivo, fopen() retorna um nulo.

Como a tabela abaixo mostra, um arquivo pode ser aberto ou em modo texto ou em modo binário. No modo texto, as seqüências de retorno de carro e alimentação de formulários (CR/LF, que significa Carriage Return/Line Feed, ou simplesmente ENTER) são transformadas em seqüências de novas linhas na entrada. Na saída, ocorre o inverso. Tais transformações não acontecem em um arquivo binário.

```
---------------------------------------------------------------------------
Os valores válidos para modo
---------------------------------------------------------------------------
Modo         Significado

"r"     Abre um arquivo para leitura
"w"     Cria um arquivo para escrita
"a"     Acrescenta dados para um arquivo existente
"rb"    Abre um arquivo binário para leitura
"wb"    Cria um arquivo binário para escrita
"ab"    Acrescenta dados a um arquivo binário existente
"r+"    Abre um arquivo para leitura/escrita
"w+"    Cria um arquivo para leitura/escrita
"a+"    Acrescenta dados ou cria um arquivo para leitura/escrita
"r+b"   Abre um arquivo binário para leitura/escrita
"w+b"   Cria um arquivo binário para leitura/escrita
"a+b"   Acrescenta ou cria um arquivo binário para leitura/escrita
```

```
"rt"    Abre um arquivo texto para leitura
"wt"    Cria um arquivo texto para escrita
"at"    Acrescenta dados a um arquivo texto
"r+t"   Abre um arquivo-texto para leitura/escrita
"w+t"   Cria um arquivo texto para leitura/escrita
"a+t"   Acrescenta dados ou cria um arquivo texto para leitura/escrita
---------------------------------------------------------------------------
```

Exemplo para abrir um arquivo texto:

```
FILE  *p;
p = fopen ("teste.txt", "w");
```

Frequentemente é usado algum teste para informar ao usuário, caso haja erro:

```
FILE  *out;
if((out=fopen("teste.txt","w"))==NULL)
{
   puts("\nNao foi possível abrir o arquivo.\n");
   exit(1);
}
```

A macro NULL é definida em STDIO.H. Esse método detecta qualquer erro na abertura do arquivo como um arquivo protegido contra escrita ou disco cheio, antes de tentar escrever nele. Um nulo é usado porque no ponteiro do arquivo nunca haverá aquele valor. Também introduzido por esse fragmento está outra função de biblioteca: exit(). A chamada à funçao exit() realiza o término imediato do programa. Ela tem este protótipo (encontrado em STDLIB.H):

```
void exit(int val);
```

O valor de **`val`** é retornado para o sistema operacional. Por convenção, um valor de retorno 0 significa término com sucesso para o DOS. Qualquer outro valor indica que o programa terminou por causa de algum problema (retornado para a variável ERRORLEVEL em arquivos de lote do DOS).

CUIDADO! ao utilizar os modos da função fopen() para abrir um arquivo, pois qualquer arquivo preexistente com o mesmo nome será apagado e o novo arquivo iniciado. Se não existirem arquivos com o nome, então será criado um. Se quiser acrescentar dados ao final do arquivo, deve usar o modo "a". Para se abrir um arquivo para operações de leitura é necessário que o arquivo já exista. Se ele não existir, será retornado um erro. Finalmente, se o arquivo é aberto para escrita/leitura, as operações feitas não apagarão o arquivo, se ele existir; entretanto, se não existir, será criado.

20.3 Escrevendo um caractere

A função **`putc()`** é usada para escrever caracteres para um fluxo que foi previamente aberto para escrita pela função **`fopen()`**. A função é declarada como:

```
int putc(int ch, FILE *fp);
```

onde **`fp`** é o ponteiro de arquivo retornado pela função **`fopen()`** e **`ch`**, o caractere a ser escrito. Por razões históricas, ch é chamado formalmente de int, mas somente o caractere de mais baixa ordem é usado.

Se uma operação com a função putc() for satisfatória, ela retornará o caractere escrito. Em caso de falha, um EOF é retornado. EOF (End Of File) é uma macrodefinição em STDIO.H que significa fim do arquivo.

20.4 Lendo um caractere

A função **getc()** é usada para ler caracteres do fluxo aberto em modo de leitura pela função **fopen()**. A função é declarada como:

```
int getc(FILE *fp)
```

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado pela função **fopen()**.

Por razoes históricas, a função **getc()** retorna um inteiro, porém o byte de mais alta ordem é zero.

A função **getc()** retornará uma marca **EOF** quando o fim do arquivo tiver sido encontrado ou um erro tiver ocorrido. Portanto, para ler um arquivo-texto até que a marca de fim de arquivo seja mostrada, você poderá usar o seguinte código:

```
ch=getc(fp);
while(ch!=EOF)
{
   ch=getc(fp);
}
```

20.5 Usando a função feof()

Quando um arquivo binário é aberto, é possível encontrar um valor inteiro que marca o final do arquivo (EOF, End Of File). Isso pode fazer com que, na rotina anterior, seja indicada uma condição de fim de arquivo, ainda que o final do arquivo físico não tenha sido encontrado. Para resolver esse problema, foi incluída a função feof() que é usada para determinar o final de um arquivo quando da leitura de dados binários. A função feof() tem este protótipo:

```
int feof(FILE *fp);
```

O seu protótipo está em STDIO.H. Ela retorna verdadeiro se o final do arquivo tiver sido encontrado; caso contrário, é retornado um zero. Portanto, a seguinte rotina lê um arquivo binário até que o final do arquivo seja encontrado. Este método pode ser aplicado para arquivo texto ou binário.

```
while(!feof(fp))ch=getc(fp);
```

20.6 Fechando um arquivo

A função **fclose()** é usada para fechar um fluxo que foi aberto por uma chamada à função **fopen()**. Ela escreve quaisquer dados restantes na área intermediária (fluxo) no arquivo e faz um fechamento formal em nível de sistema operacional do arquivo. Uma falha no fechamento de um arquivo leva a todo tipo de problemas, incluindo-se perda de dados, arquivos destruídos e a possibilidade de erros intermitentes no seu programa. Uma chamada à funçao fclose() também libera os blocos de controle de arquivo (FCB) associados ao fluxo e deixa-os disponíveis para reutilização.

O sistema operacional limita o número de arquivos abertos num dado momento. Para abrir outros arquivos, basta fechar aqueles que não estiverem em uso. Isto também pode ser configurado no autoexec.bat através do ajuste de FILES.

A função fclose() é declarada como:

```
int fclose(FILE*fp);
```

onde fp é o ponteiro do arquivo retornado pela chamada à função fopen(). Um valor de retorno igual a zero significa uma operação de fechamento com sucesso; qualquer outro valor indica um erro. É possível usar a função padrão ferror() (discutida a seguir) para determinar e reportar quaisquer problemas. Geralmente, a função fclose() falhará é quando uma uniadde de mídia (disquete ou pendrive) não estiver mais disponível no sistema (devido a remoção prematura) ou falta de espaço livre na mídia.

A função **ferror()** é usada para determinar se uma operação em um arquivo produziu algum erro. A função ferror() tem o seguinte protótipo:

```
int ferror(FILE*fp);
```

onde fp é um ponteiro válido para um arquivo. ferror() retorna verdadeiro se um erro ocorreu durante a última operação com o arquivo e falso, caso contrário. Uma vez que cada operação em arquivo determina uma condição de erro, a função ferror() deve ser chamada imediatamente após cada operação com o arquivo; caso contrário, um erro pode ser perdido. O protótipo de função ferror() está no arquivo STDIO.H.

A função **rewind()** recolocará o localizador de posição do arquivo no início do arquivo especificado como seu argumento. O seu protótipo é:

```
void rewind(FILE*fp);
```

onde fp é um ponteiro de arquivo. O protótipo para a funçao rewind() está no arquivo STDIO.H.

Exemplos:

```
/* Grava em disco dados digitados no teclado até receber $

   Utilização: programa arquivo.txt w
*/

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{  FILE *fp;
   char ch;
   if(argc!=3)
   {  puts("AJUDA: digite apos o nome do programa o arquivo e o modo!");
      exit(1);
   }
   // Experimente 'wb' e 'wt' para testar newline (ENTER)
   if((fp=fopen(argv[1], argv[2]))==NULL)
   {  puts("O arquivo nao pode ser aberto!");
      exit(1);
   }
   do
   {  ch=getchar(); // lê caracteres do teclado no buffer até newline
      if(EOF==putc(ch, fp))
      {  puts("Erro ao escrever no arquivo!");
         break;
      }
```

```
   } while (ch!='$');
   fclose(fp);
   return 0;
}

// ##########################################################################

// Le arquivos e exibe-os na tela

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{  FILE *fp;
   char ch;
   if(argc!=3)
   {  puts("Você se esqueceu de informar o nome do arquivo e o modo!");
      exit(1);
   }
   // Experimente 'rb' e 'rt' para testar newline
   if((fp=fopen(argv[1], argv[2]))==NULL)
   {  puts("O arquivo nao pode ser aberto!");
      exit(1);
   }
   ch= getc(fp); /* lê um caractere */
   while(ch!=EOF)
   {  putchar(ch); /* imprime na tela */
      ch=getc(fp);
   }
   return 0;
}
```

getw() e putw():

Funcionam exatamente como putc() e getc() só que em vez de ler ou escrever um único caractere, lêem ou escrevem um inteiro. Protótipos:

```
#include <stdio.h>

void main()
{
   int putw(int i, FILE *fp);
   int getw(FILE *fp);
}
```

Exemplo:

```
// escreve o inteiro 100 no arquivo apontado por saida.
putw(100,saida);
```

fgets() e fputs():

Funcionam como gets() e puts() só que lêem e escrevem em fluxos. Protótipos:

```
#include <stdio.h>

void main()
{
```

```
   char *fputs(char *str, FILE *fp);
   char *fgets(char *str, int tamanho, FILE *fp);
}
```

ATENÇÃO: fgets() lê uma string de um fluxo especificado até que um newline seja lido OU tamanho-1 caracteres tenham sido lidos. Se um newline é lido, ele fará parte da string (diferente de gets()). A string resultante termina com um nulo.

fread() e fwrite():

Lêem e escrevem blocos de dados em fluxos. Protótipos:

```
#include <stdio.h>

void main()
{
   unsigned fread(void *buffer, int num_bytes, int count, FILE *fp);
   unsigned fwrite(void *buffer, int num_bytes, int count, FILE *fp);
}
```

Para **`fread()`**, buffer é um ponteiro para uma área da memória que receberá os dados lidos do arquivo. Já para **`fwrite()`**, buffer é um ponteiro para uma área da memória onde se encontram os dados a serem escritos no arquivo. Buffer usualmente aponta para uma matriz ou estrutura. O número de bytes a ser lido ou escrito é especificado por **`num_bytes`**.

O argumento count determina quantos ítens (cada um tendo num_bytes de tamanho) serão lidos ou escritos. O argumento fp é um ponteiro para um arquivo de um fluxo previamente aberto por fopen(). A função fread() retorna o número de itens lidos, que pode ser menor que count caso o final de arquivo (EOF) seja encontrado ou ocorra um erro. A função fwrite() retorna o número de itens escritos, que será igual a count exceto na ocorrência de um erro de escrita.

Quando o arquivo for aberto para dados binários, fread() e fwrite() podem ler e escrever qualquer tipo de informaçao. O programa a seguir escreve um float em um arquivo de disco chamado teste.dat:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

main()
{  FILE *fp;
   float f=M_PI;

   if((fp=fopen("teste.dat","wb"))==NULL)
   {  puts("Nao posso abrir arquivo!");
      exit(1);
   }
   if(fwrite(&f, sizeof(float), 1, fp)!=1)
      puts("Erro escrevendo no arquivo!");
   fclose(fp);
   return 0;
}
```

Como o programa anterior ilustra, a área intermediária de armazenamento pode ser (e frequentemente é) simplesmente uma variável.

Uma das aplicaçoes mais úteis de fread() e fwrite() é o armazenamento e leitura rápidos de matrizes e estruturas (estrutura é uma entidade lógica a ser vista adiante) em disco:

```
/* escreve uma matriz em disco */
#include<stdio.h>
#include<stdlib.h>

main()
{  FILE *fp;
   float exemplo[10][10];
   int i,j;
   if((fp=fopen("exemplo.dat","wb"))==NULL)
   {  puts("Nao posso abrir arquivo!");
      exit(1);
   }
   for(i=0; i<10; i++)
   {  for(j=0; j<10; j++)
      {  /* lei de formaçao dos elementos da matriz */
         exemplo[i][j] = (float) i+j;
      }
   }
   /* o código a seguir grava a matriz inteira em um único passo: */
   if(fwrite(exemplo, sizeof(exemplo), 1, fp)!=1)
   {  puts("Erro ao escrever arquivo!");
      exit(1);
   }
   fclose(fp);
   return 0;
}


/* lê uma matriz em disco */
#include<stdio.h>
#include<stdlib.h>

main()
{  FILE *fp;
   float exemplo[10][10];
   int i,j;
   if((fp=fopen("exemplo.dat","rb"))==NULL)
   {  puts("Nao posso abrir arquivo!");
      exit(1);
   }
   /* o código a seguir lê a matriz inteira em um único passo: */
   if(fread(exemplo, sizeof(exemplo), 1, fp)!=1)
   {  puts("Erro ao ler arquivo!");
      exit(1);
   }
   for(i=0; i<10; i++)
   {  for(j=0; j<10; j++)
         printf("%3.1f ", exemplo[i][j]);
      printf("\n");
   }
   fclose(fp);
   return 0;
}
```

20.7 Acesso randômico a arquivos

A função fseek() indica o localizador de posição do arquivo. Protótipo:

```
#include <stdio.h>

main()
{
   int fseek(FILE *fp, long numbytes, int origem);
}
```

Onde:

- fp é um ponteiro para o arquivo retornado por uma chamada à fopen().
- numbytes (long int) é o número de bytes a partir da origem até a posição corrente.
- origem é uma das seguintes macros definidas em stdio.h:

Origem	Nome da Macro	Valor numérico
começo do arquivo	SEEK_SET	0
posiçao corrente	SEEK_CUR	1
fim do arquivo	SEEK_END	2

Portanto, para procurar numbytes do começo do arquivo, origem deve ser SEEK_SET. Para procurar da posição corrente em diante, origem é SEEK_CUR. Para procurar numbytes do final do arquivo de trás para diante, origem é SEEK_END.

O seguinte trecho de código lê o 235º byte do arquivo chamado test:

```
   FILE *fp;
   char ch;
   if((fp=fopen("teste","rb"))==NULL)
   {  puts("Nao posso abrir arquivo!");
      exit(1);
   }
   fseek(fp,234,SEEK_SET);
   ch=getc(fp);   /* lê um caractere na posiçao 235° */
```

A função fseek() retorna zero se houve sucesso ou um valor nao-zero se houve falha no posicionamento do localizador de posiçao do arquivo.

```
/* Utilitario de vizualizacao de disco usando fseek()

   Visualiza setores de 128 de bytes do disco e apresenta
   em ASCII e hexadecimal digite -1 para sair do programa.
*/


#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define TAMANHO 128

char buf[TAMANHO];

void display();

main(int argc, char *argv[])
```

```
{  FILE *fp;
   long int setor, numlido;
   if(argc!=2)
   {  puts("Uso: tmp nome_do_arquivo");
      exit(1);
   }
   if((fp=fopen(argv[1],"rb"))==NULL)
   {  puts("Nao posso abrir arquivo!");
      exit(1);
   }
   for(;;)
   {  printf("Informe o setor (-1 p/ terminar): ");
      scanf("%ld",&setor);
      if(setor<0) break;
      if(fseek(fp, setor*TAMANHO , SEEK_SET))
      {  puts("Erro de busca!");
         exit(1);
      }
      if((numlido=fread(buf,1,TAMANHO,fp)) != TAMANHO)
      {  puts("EOF encontrado!");
      }
      display(numlido);
   }
   return 0;
}


void display(long int numlido)
{  long int i,j;
   for(i=0; i<=numlido/16 ; i++)
   {  /* controla a indexaçao de linhas: 0 a 8 (128/16=8) */
      for(j=0; j<16; j++)
         printf("%2X ", buf[i*16+j]); /* imprime 16 col. em hexa */
      printf("  "); /* separa mostrador hexa do ascii */
      for(j=0; j<16; j++)
      {  /* imprime 16 col. em ascii: (só imprimíveis) */
         if(isprint(buf[i*16+j]))
            printf("%c",buf[i*16+j]);
         else
            printf(".");
      }
      printf("\n");
   }
}
```

20.8 Fluxo Padrão

Toda vez que um programa começa a execução, são abertos 5 fluxos por padrão:

stdin (aponta p/ o teclado se nao redirecionado pelo DOS. Ex: MORE < FILE.TXT)
stdout (aponta p/ a tela se nao redirecionado pelo DOS. Ex : DIR > PRN)
stderr (recebe mensagens de erro - aponta para a tela)
stdprn (aponta p/ a impressora)
stdaux (aponta p/ a porta serial)

Para entender o funcionamento destes fluxos, note que putchar() é definida em stdio.h como:

```
#define putchar(c) putc(c,stdout)
```

e a funçao getchar() é definida como

```
#define getchar() getc(stdin)
```

Ou seja, estes fluxos permitem serem lidos e escritos como se fossem fluxos de arquivos. Toda entrada de dado de um program é feita por stdin e toda saída por stdout:

```
/* Localiza uma palavra especificada pela linha de comando em um arquivo
   redirecionado para stdin, mostrando a linha e seu número.
   No prompt de comando, digite: tmp argv < tmp.c
*/
#include<string.h>
#include<stdio.h>

main(int argc, char *argv[])
{  char string[128];
   int line=0;
   while(fgets(string, sizeof(string),stdin))
   {  ++line;
      if(strstr(string, argv[1]))
         printf("%02d %s",line,string);
   }
}
```

O fluxo stderr é usado para mostrar mensagens de erro na tela, quando a saída do programa esta redirecionada para outro dispositivo que não seja a tela:

```
#include <stdio.h>
main(int argc, char *argv[])
{  FILE *fp;
   int count;
   char letter;
   if(argc!=2)
   {  puts("Digite o nome do arquivo!");
      exit(1);
   }
   if(!(fp = fopen(argv[1],"w")))
   {  fputs("Erro na abertura do arquivo",stderr);
      /* outra opçao: puts("Erro na abertura do arquivo"); */
      exit(1);
   }
   else
      for(letter='A';letter<='Z'; letter++)
         putc(letter, fp);
   fclose(fp);
}
```

Se a saída deste programa (stdout) for redirecionada para algum outro arquivo, a mensagem de erro forçosamente aparecerá na tela porque estamos escrevendo em stderr.

fprintf() e fscanf():

Comportam-se como printf() e scanf() só que escrevem e lêem de arquivos de disco. Todos os códigos de formato e modificadores são os mesmos. Protótipo:

```
#include <stdio.h>
main()
{
   int fprintf(FILE *fp, char *string_de_controle, lista_de_argumentos);
   int fscanf(FILE *fp, char *string_de_controle, lista_de_argumentos);
}
```

Embora fprintf() e fscanf() sejam a maneira mais fácil de ler e escrever tipos de dados nos mais diversos formatos, elas não são as mais eficientes em termos de tamanho de código resultante e velocidade. Quando o formato de leitura e escrita for de importância secundária, deve-se dar preferência a fread() e fwrite().

```
/* imprime os quadrados de 0 a 10 no arquivo quad.dat */
#include<stdio.h>
#include<stdlib.h>

main()
{  int i;
   FILE *out;
   if((out=fopen("quad.dat","wt"))==NULL)
   {  puts("Nao posso abrir arquivo!");
      exit(1);
   }
   for (i=0; i<=10; i++)
      fprintf(out,"%d   %d\n", i , i*i);
   fclose(out);
}
```

Apagando arquivos: remove()

```
int remove(char *nome_arquivo);
```

Retorna zero em caso de successo e não zero se falhar.

20.9 Exemplo de utilização de arquivo texto

Salve este trecho como 'nomes.txt'

```
---inicio, nao copie esta linha
5
Joao da Silva # 26
Antonio Silvestre # 35
Felipe Barreto # 21
Marcos Miranda # 20
Xavier Goncalves # 70
---fim, nao copie esta linha
```

```
// Este programa le um arquivo 'nomes.txt' e ordena os registros por nome
// em memoria, gravando a seguir em outro arquivo 'arq-ord.txt'.

#include <stdio.h>

typedef struct _amigo
{  char  nome[100];
   int   telefone;
} Amigo;

typedef struct _agenda
{  Amigo *amigo;
   int num_amigos;
} Agenda;

Agenda LeAgenda(char *nomearq);
void   OrdenaAgenda(Agenda a);
void   GravaAgenda(Agenda a, char *nomearq);
void   DestroiAgenda(Agenda a);
```

```
Agenda LeAgenda(char *nomearq)
{  Agenda a;
   FILE *aparq;
   int i;
   char linha[2000], *pos;

   //abre arquivo texto para leitura
   aparq = fopen(nomearq, "r");
   // Le cadeia de caracteres ate encontrar \n
   fgets(linha,199,aparq);
   sscanf(linha,"%d",&a.num_amigos);
   a.amigo = (Amigo *) calloc(a.num_amigos,sizeof(Amigo));

   if(a.amigo != NULL)
   {  for(i=0; i <a.num_amigos; i++)
      {  fgets(linha,199,aparq);
         pos=strchr(linha,'#');
         strncpy(a.amigo[i].nome,linha,pos-linha-1);
         sscanf(pos+1,"%d",&a.amigo[i].telefone);
      }
   }
   // fecha o arquivo
   fclose(aparq);
   return(a);
}

void OrdenaAgenda(Agenda a)
{  int i,j,jm;
   Amigo aux;
   for(j=0; j<a.num_amigos-1; j++)
   {  jm = j;
      for (i=j+1; i < a.num_amigos; i++)
      {  if(strcmp(a.amigo[i].nome,a.amigo[jm].nome) < 0)
         {  jm = i; }
      }
      if (j != jm)
      {  aux = a.amigo[j];
         a.amigo[j] = a.amigo[jm];
         a.amigo[jm] = aux;
      }
   }
}

void GravaAgenda(Agenda a, char *nomearq)
{  int i;
   FILE *aparq;
   // cria novo arquivo texto para escrita
   aparq = fopen(nomearq,"w");
   // escreve no arquivo
   fprintf(aparq,"%d\n",a.num_amigos);
   for (i=0; i < a.num_amigos; i++)
      fprintf(aparq,"%s # %d\n",a.amigo[i].nome,a.amigo[i].telefone);
   // fecha arquivo
   fclose(aparq);
}

void DestroiAgenda(Agenda a)
{  if (a.amigo != NULL)
      free(a.amigo);
}

int main()
{  Agenda a;
```

```
   a = LeAgenda("nomes.txt");
   OrdenaAgenda(a);
   GravaAgenda(a,"arq-ord.txt");
   DestroiAgenda(a);
   return(0);
}
```

20.10 Exercício Proposto:

20.10.1) Faça uma agenda que armazene nome, endereço, telefone e e-mail dos contatos. O programa permitirá que o usuário insira, remova e procure seus contatos.

21 – Sistemas

Um sistema é um conjunto de programas que compartilham uma ou mais bibliotecas de funções. Quando projetamos um sistema, cada estrutura abstrata e as funções que realizam operações com esta estrutura devem ser declaradas em arquivos com extensões .h e .c separados dos arquivos de programa. Por exemplo, nos arquivos contorno.h e matriz.h nós declaramos estruturas e protótipos das funções.

```
//=== contorno.h ===

#ifndef CONTORNO_H
#define CONTORNO_H 1

#include <malloc.h>

typedef struct _ponto
{  float x,y;
} Ponto;

typedef struct _contorno
{  Ponto *p; // vetor de pontos
   int n;    // numero de pontos
} Contorno;

// Aloca espaco para contorno com N pontos
Contorno CriaContorno(int n);

// Desaloca espaco do contorno
void DestroiContorno(Contorno c);

/* Aqui voce pode acrescentar os prototipos
   de outras funcoes que manipulam contorno
*/

#endif
```

```
//=== matriz.h ===

#ifndef MATRIZ_H
#define MATRIZ_H 1

#include <malloc.h>

typedef struct _matriz
{  // elementos da matriz
   float **elem;
   // numero de linhas e colunas
   int nlin, ncol;
} Matriz;

// Aloca espaco para a matriz nlin X ncol
Matriz CriaMatriz(int nlin, int ncol);
// Desaloca espaco
void DestroiMatriz(Matriz m);

/* Aqui voce pode acrescentar os prototipos
   de outras funcoes que manipulam Matriz */

#endif
```

Nos arquivos contorno.c e matriz.c incluímos os respectivos arquivos .h e declaramos os escopos dessas funções:

```
//=== contorno.c ===

#include "contorno.h"

Contorno CriaContorno(int n)
{  Contorno c;
   c.n = n;
   c.p = (Ponto *)calloc(n,sizeof(Ponto));
   return(c);
}

void DestroiContorno(Contorno c)
{  if(c.p != NULL)
   {  free(c.p);
      c.p = NULL;
   }
}

/* Aqui voce acrescenta os escopos das
outras funcoes que manipulam Contorno */
```

```
//=== matriz.c ===

#include "matriz.h"

Matriz CriaMatriz(int nlin, int ncol)
{  Matriz m;
   int i;
   m.nlin = nlin;
   m.ncol = ncol;
   m.elem = (float **)calloc(nlin, sizeof(float *));
   if (m.elem != NULL)
     for (i=0; i<nlin; i++)
      m.elem[i] = (float *)calloc(ncol, sizeof(float));
   return(m);
}

void DestroiMatriz(Matriz m)
{  int i;
   if(m.elem != NULL)
   {  for (i=0; i<m.nlin; i++)
         free(m.elem[i]);
      free(m.elem);
      m.elem = NULL;
   }
}

/* Aqui voce acrescenta os escopos das outras
   funcoes que manipulam Matriz */
```

Para organizar o sistema podemos criar uma estrutura de diretórios com raiz **mc102**, por exemplo, colocando os arquivos com extensão .h em **mc102/include** e os arquivos com com extensão .c em **mc102/src**. Como um arquivo com a extensão .c não possui a função main, ele não é considerado um programa. Seu código, após compilado, tem extensão .o e é denominado **código objeto** (não executável). Os respectivos arquivos com extensão .o podem ser colocados em **mc102/obj**. Para que as estruturas e funções desses arquivos sejam disponibilizadas para uso por programas, os arquivos objeto devem ser “ligados” em um arquivo com extensão .a (ex: arquivo biblioteca de funções). O script **Makefile** abaixo realiza esta tarefa colocando a biblioteca libmc102.a no diretório **mc102/lib**. O Makefile fica no diretório mc102.

```
LIB=./lib
INCLUDE=./include
BIN=./
SRC=./src
OBJ=./obj

#FLAGS= -g -Wall
FLAGS= -O3 -Wall

libmc102: $(LIB)/libmc102.a
echo "libmc102.a construida com sucesso..."

$(LIB)/libmc102.a: \
$(OBJ)/contorno.o \
$(OBJ)/matriz.o

ar csr $(LIB)/libmc102.a \
$(OBJ)/contorno.o \
$(OBJ)/matriz.o

$(OBJ)/contorno.o: $(SRC)/contorno.c
gcc $(FLAGS) -c $(SRC)/contorno.c -I$(INCLUDE) \
-o $(OBJ)/contorno.o

$(OBJ)/matriz.o: $(SRC)/matriz.c
gcc $(FLAGS) -c $(SRC)/matriz.c -I$(INCLUDE) \
-o $(OBJ)/matriz.o

apaga:
rm $(OBJ)/*.o;

apagatudo:
rm $(OBJ)/*.o; rm $(LIB)/*.a
```

21.1 Programas

Todo programa que quiser usar as estruturas e funções da biblioteca libmc102.a devem incluir os arquivos .h da biblioteca e devem ser compilados com o comando abaixo. Suponha por exemplo, que o programa prog.c está no mesmo nível do diretório mc102.

```
//=== prog.c ===

#include "contorno.h"
#include "matriz.h"

int main()
{  Contorno.c
   Matriz m;
```

```
   c = CriaContorno(100);
   m = CriaMatriz(5,9);

   DestroiContorno(c);
   DestroiMatriz(m);
   return(0);
}
```

Comando para compilação

```
gcc prog.c -I./mc102/include -L./mc102/lib -o prog -lmc102
```

21.2 Argumentos de Programa

Em aula anterior vimos a utilização de nome de arquivo fixo no programa, por exemplo:

```
int main()
{  Agenda a;

   a = LeAgenda("arquivo.txt");
   OrdenaAgenda(a);
   GravaAgenda(a,"arq_ord.txt");
   DestroiAgenda(a);
   return(0);
}
```

Porém seria mais conveniente se esses nomes fossem passados como argumentos do programa na linha de execução (ex: **programa arquivo.txt arq_ord.txt**). Isto é possível, pois a função main pode ser declarada com dois argumentos como veremos no exemplo a seguir. O valor de **argc** é o número de argumentos em argv e **argv** é um vetor de strings, onde cada string contém um argumento do programa (ex: um número inteiro; um número real; um caractere ou uma cadeia de caracteres). Ajustando o programa acima, temos:

```
int main(int argc, char **argv)
{  Agenda a;

   if (argc != 3)
   {  printf(uso: %s <entrada> <saida\n", argv[0]);
      exit(-1);
   }
   a = LeAgenda(argv[1]);
   OrdenaAgenda(a);
   GravaAgenda(a,argcv[2]);
   DestroiAgenda(a);
   return(0);
}
```

Em casos de argumentos inteiros e reais, as funções **atoi** e **atof** podem ser usadas para converter strings nesses valores, respectivamente.

```
#include “contorno.h”
#include “matriz.h”
int main(int argc, char **argv)
{  Contorno c;
   Matriz m;

   if (argc != 4)
   {    printf("uso:  %s  <numero  de  pontos>  <numero  de  linhas>  <numro  de
colunas>\n", argv[0]);
      exit(-1);
   }
```

```
    c = CriaContorno(atoi(argv[1]);
    m = CriaMatriz(atoi(argv[2],atoi(argv[3]));

    DestroiContorno(c);
    DestroiMatriz(m);
    return(0);
}
```

Bibliografia consultada

CELES, W.; RANGEL, J.L. “Listas Encadeadas”, PUC-Rio.

FALCÃO, Alexandre Xavier. “Notas de aula – MC102”. IC/UNICAMP, 2008

FORBELLONE, André Luiz Villar, Eberspächer, Henri Frederico; “Lógica de Programação, A Construção de Algoritmos e Estrutura de Dados”. – 2ª edição, MAKRON. – São Paulo.

GOTTFRIED, Byron S. “Programando em C”; Ed. Makron Books, 1993, SP.

MANSSOUR, Isabel. “Linguagem de Programação C – Ponteiros”, 2008.

MANZANO, José Augusto N. G., OLIVEIRA, Jayr Figueiredo de; “Algoritmos – Lógica para Desenvolvimento de Programação”. 3ª edição, EDITORA ÉRICA. – São Paulo.

MAYER, Roberto C.; “Linguagem C ANSI – Guia do usuário”, Ed. McGrawHill, SP, 1989.

MEIRELLES, Fernando de Souza. "Informática: Novas aplicações com microcomputadores"; 2a Ed. São Paulo: Makron Books, 1994.

MORAES, Paulo S. “Curso de Lógica de Programação”. DSC/Unicamp, 2000.

SCHILDT, Herbert. “C, Completo e Total”. 3a Ed. São Paulo: Makron Books, 1996.

TANENBAUM, Andrew S. “Sistemas Operacionais Modernos”. 2ª Ed. Cap.10

ANEXO A – Fluxogramas

Introdução: **Um pouco de história**

O algoritmo nasceu da mente brilhante do matemático Leonard Euler, nascido no ano de 1707, na cidade de Basiléia, Suíça. Seu pai queria que fosse pastor, mas sua vocação estava na matemática. Estudava cada problema que lhe aparecesse, como: a navegação, finanças, irrigação entre outros. Para cada problema, Euler criava novos cálculos. O algoritmo surgiu com o cálculo do comportamento da lua, criou uma teoria chamada "o problema dos 3 corpos", o algoritmo permitia que Euler encontrasse um valor aproximado para a posição da lua, esses estudos vieram à favorecer a navegação da época, num período onde não se dispunham de muitos instrumentos de navegação.

Também utilizado por Alan Turing. O algoritmo é um conjunto de instruções para executar uma ou várias tarefas. Muito antes dos computadores, mais precisamente no ano de 1936, Alan Turing, fundamentou o conceito de algoritmo, constituídos de seqüências e passos para tomada de decisões, não se trata da linguagem de programação mas serve como parâmetro para o desenvolvimento da programação.

A mesma tarefa pode ser realizada por diferentes algoritmos constituídos de instruções diferenciadas utilizando-se de mais tempo ou menos tempo dependendo de sua estrutura.

Com o emprego da informática, o algoritmo passa a ser o instrumento básico para todas as linguagens no desenvolvimento de programas para os computadores.

Para a construção de um algoritmo, é necessário aprender a pensar, utilizar-se da lógica, seus passos devem permitir uma única interpretação correta para que se realize a tarefa sem erros ou ambigüidades.

Toda a estrutura obedece uma formatação, para uma distinção das seqüências lógica, podendo calcular, armazenar e ser constituída de rotinas de conferência até a realização completa da tarefa executada pelo usuário atingindo seu objetivo.

Entendendo as nomenclaturas

Os termos: fluxograma, diagrama de blocos ou algoritmos, são muito utilizados pelos profissionais de processamento de dados, apesar de constituídos de símbolos que representam a linha de raciocínio lógico, elas têm significados divergentes.

Entrelaçado à lógica, o algoritmo pode ser definido como regras formais para obtenção de um resultado, é também conhecido como uma seqüência de passos ou etapas visando chegar a um objetivo. O diagrama de blocos tem como propósito, descrever o método e a seqüência do processo, já o fluxograma é a ferramenta usada com a finalidade de descrever o fluxo. Usa símbolos convencionais, permitindo poucas variações.

TIPOS DE FLUXOGRAMA

Existe uma infinidade de diagramas para diversos tipos de relações:

Diagrama de auditoria	Este diagrama é utilizado para documentar e analisar processos financeiros.
Fluxograma básico	Serve para documentar procedimentos, fluxo de trabalho ou de informações.
Diagrama de causa e efeito	Muito utilizado para complemento de implantação de Sistema de Qualidade, conhecido também como diagrama de Ishikawa.
Fluxograma multifuncional	Utilizado para apresentar o processo de unidades organizacionais e seus respectivos responsáveis pelo processo.
Diagrama de fluxo de dados	Trata-se de fluxo lógico de dados, utilizado para documentar procedimento de informação de dados e armazenamento
Diagrama SDL	Utilizado na documentação e análises de sistemas de rede de comunicação e telecomunicação por meio de linguagem.
Diagrama TQM	Serve para gerenciamento de qualidade total, melhoramento contínuo e soluções de problemas voltados à qualidade.
Diagrama de Fluxo de Trabalho	Automação de processos industriais, contábeis, entre outros.